Universidad de los Andes
Facultad de Ingeniería

**Departamento de Ingeniería de Sistemas y Computación**
**Estructuras de Datos y Algoritmos**
**ISIS-1225**

ABET Engineering Accreditation Commission

# ANÁLISIS DEL RETO 4

*Valeria Gutieerez, código 1, email 1*

*Andrea Aroca, 202320457, c.arocad@uniandes.edu.co*

*Juan David Calderón, 202320117, jd.calderong12@uniandes.edu.co*

## Carga de Datos

### Descripción

```
def new_data_structs():
        'vuelos':mp.newMap(numelements=430
                           ,
                                maptype='PROBING'),
        'aeropuertos_mapa':mp.newMap(numelements=430
                           ,
                                maptype='PROBING'),
        'aviacion_carga_distancia': gr.newGraph(datastructure='ADJ_LIST',
                                directed=True,
                                size=430,
                               ),

        'aviacion_carga_tiempo': gr.newGraph(datastructure='ADJ_LIST',
                                directed=True,
                                size=430,
                                ),
        'aviacion_comercial_distancia': gr.newGraph(datastructure='ADJ_LIST',
                                directed=True,
                                size=430,
                                ),
        'aviacion_comercial_tiempo':  gr.newGraph(datastructure='ADJ_LIST',
                                directed=True,
                                size=430,
                                ),
        'militar_distancia':  gr.newGraph(datastructure='ADJ_LIST',
                                directed=True,
                                size=430,
                                ),           'aviacion_comercial_tiempo':  gr.newGraph(datastruct
                                directed=True,
                                size=430,
                                ),
        'militar_tiempo':  gr.newGraph(datastructure='ADJ_LIST',
                                directed=True,
                                size=430,
                                ),           'aviacion_comercial_tiempo':  gr.newGraph(datastruct
                                directed=True,
                                size=430,
                                ),
```

```python
def add_aeropuerto(analyzer, aeropuerto):
    aeropuertos_mapa = analyzer['aeropuertos_mapa']
    id_aeropuerto = aeropuerto['ICAO']
    aeropuerto['Concurrencia carga'] = 0
    aeropuerto['Concurrencia comercial'] = 0
    aeropuerto['Concurrencia militar'] = 0
    mp.put(aeropuertos_mapa, id_aeropuerto, aeropuerto)


def carga_grafos_mapa_vuelos(analyzer, vuelos):
    #función que carga todos los grafos y el mapa de vuelos
    grafo_carga_d= analyzer['aviacion_carga_distancia']
    grafo_comercial_d= analyzer['aviacion_comercial_distancia']
    grafo_militar_d= analyzer['militar_distancia']
    grafo_carga_t=analyzer['aviacion_carga_tiempo']
    grafo_comercial_t= analyzer['aviacion_comercial_tiempo']
    grafo_militar_t= analyzer['militar_tiempo']
    tipo_vuelo= vuelos['TIPO_VUELO']
    if tipo_vuelo=='AVIACION_CARGA':
       add_vertices(analyzer, vuelos, grafo_carga_t, 'tiempo','AVIACION_CARGA')
       add_vertices(analyzer, vuelos, grafo_carga_d, 'distancia', 'AVIACION_CARGA')

    elif tipo_vuelo=='AVIACION_COMERCIAL':
       add_vertices(analyzer, vuelos, grafo_comercial_d, 'distancia', 'AVIACION_COMERCIAL')
       add_vertices(analyzer, vuelos, grafo_comercial_t, 'tiempo', 'AVIACION_COMERCIAL')

    elif tipo_vuelo=='MILITAR':
      add_vertices(analyzer, vuelos, grafo_militar_d, 'distancia', 'MILITAR')
      add_vertices(analyzer, vuelos, grafo_militar_t, 'tiempo', 'MILITAR')
    add_vuelo(analyzer, vuelos)
```

```python
def add_vertices(analyzer, vuelos, grafo, tipo, tipo_vuelo):
    if vuelos['TIPO_VUELO'] != tipo_vuelo:
        return
    else:
        origen = vuelos['ORIGEN']
        destino = vuelos['DESTINO']

        if not gr.containsVertex(grafo, origen):
            gr.insertVertex(grafo, origen)
        if not gr.containsVertex(grafo, destino):
            gr.insertVertex(grafo, destino)

        peso = calc_arco(analyzer['aeropuertos_mapa'], vuelos, origen, destino, tipo)
        if gr.getEdge(grafo, origen, destino) is None:
            gr.addEdge(grafo, origen, destino, peso)


def calc_arco(mapa_aeropuertos, vuelos, origen, destino, tipo):
    #función que adiciona arcos, si es un grafo de distancia utiliza la fórmula Harvesine y si es
    if tipo=='distancia':
        R=6372.8
        pareja_origen= mp.get(mapa_aeropuertos, origen)
        datos_origen=me.getValue(pareja_origen)
        pareja_destino= mp.get(mapa_aeropuertos, destino)
        datos_destino=me.getValue(pareja_destino)
        lat_o= math.radians(float((datos_origen['LATITUD']).replace(',', '.')))
        lon_o= math.radians(float((datos_origen['LONGITUD']).replace(',', '.')))
        lat_d=  math.radians(float((datos_destino['LATITUD']).replace(',', '.')))
        lon_d=  math.radians(float((datos_destino['LONGITUD']).replace(',', '.')))
        dlat = lat_o - lat_d
        dlon = lon_o - lon_d
        a = math.sin(dlat / 2)**2 + math.cos(lat_o) * math.cos(lat_d) * math.sin(dlon / 2)**2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
        distance = R * c
        return distance
    elif tipo=='tiempo':
        tiempo= int(vuelos['TIEMPO_VUELO'])
```

```python
def calcular_concurrencia_por_categoria(analyzer):
    grafo_carga_d = analyzer['aviacion_carga_distancia']
    grafo_comercial_d = analyzer['aviacion_comercial_distancia']
    grafo_militar_d = analyzer['militar_distancia']
    grafo_carga_t = analyzer['aviacion_carga_tiempo']
    grafo_comercial_t = analyzer['aviacion_comercial_tiempo']
    grafo_militar_t = analyzer['militar_tiempo']
    aeropuertos_mapa = analyzer['aeropuertos_mapa']

    # Calcular concurrencia para aviación de carga (tiempo y distancia)
    for grafo in [grafo_carga_d, grafo_carga_t]:
        vertices = gr.vertices(grafo)
        for vertice in lt.iterator(vertices):
            concurrencia = gr.degree(grafo, vertice)
            aeropuerto = me.getValue(mp.get(aeropuertos_mapa, vertice))
            aeropuerto['Concurrencia carga'] += concurrencia

    # Calcular concurrencia para aviación comercial (tiempo y distancia)
    for grafo in [grafo_comercial_d, grafo_comercial_t]:
        vertices = gr.vertices(grafo)
        for vertice in lt.iterator(vertices):
            concurrencia = gr.degree(grafo, vertice)
            aeropuerto = me.getValue(mp.get(aeropuertos_mapa, vertice))
            aeropuerto['Concurrencia comercial'] += concurrencia

    # Calcular concurrencia para aviación militar (tiempo y distancia)
    for grafo in [grafo_militar_d, grafo_militar_t]:
        vertices = gr.vertices(grafo)
        for vertice in lt.iterator(vertices):
            concurrencia = gr.degree(grafo, vertice)
            aeropuerto = me.getValue(mp.get(aeropuertos_mapa, vertice))
            aeropuerto['Concurrencia militar'] += concurrencia
```

```python
def add_vuelo(analyzer, vuelos):
    #función que crea el mapa de vuelos
    mapa_vuelos= analyzer['vuelos']
    nombre_vuelo= formato_id(vuelos)
    mp.put(mapa_vuelos, nombre_vuelo, vuelos)

def formato_id(vuelos):
    #función que formatea los ids del mapa de vuelos
    nombre= vuelos['ORIGEN']
    nombre= nombre +'/' + vuelos['DESTINO']
    return nombre

def reporte_de_Carga(analyzer):
    #Función que recolecta toda la informacion que se requiere para la impresion de la carga
    aeropuertos_cargados= analyzer['aeropuertos_mapa']
    total_aeropuertos_cargados= lt.size(mp.keySet(aeropuertos_cargados))
    vuelos_cargados= analyzer['vuelos']
    total_vuelos_cargados= lt.size(mp.keySet(vuelos_cargados))

    arbol_comercial= analyzer['aviacion_comercial_distancia']
    listas_comercial=listas(arbol_comercial, aeropuertos_cargados, 'Concurrencia comercial')

    arbol_carga= analyzer['aviacion_carga_distancia']
    listas_carga=listas(arbol_carga, aeropuertos_cargados, 'Concurrencia carga')

    arbol_militar= analyzer['militar_distancia']
    listas_militar=listas(arbol_militar, aeropuertos_cargados, 'Concurrencia militar')

    return total_aeropuertos_cargados, total_vuelos_cargados, listas_comercial, listas_carga, listas_militar
```

```python
def listas(arbol, aeropuertos_cargados, categoria):
    #Función que organiza por primeros 5 y últimos 5
    lista_vertices= gr.vertices(arbol)
    lista_orden= lt.newList('ARRAY_LIST')
    for i in lt.iterator(lista_vertices):
        diccionarioi= me.getValue(mp.get(aeropuertos_cargados, i))
        elementos= i + '/' + str(diccionarioi[categoria])
        sin_cero=elementos.split('/')
        if int(sin_cero[1]) != 0:
            lt.addLast(lista_orden, elementos)
    merg.sort(lista_orden, degrees_cmp)
    lista_primeros= lt.subList(lista_orden, 1, 5)
    lista_ultimos= lt.subList(lista_orden, lt.size(lista_orden)-4, 5)
    lt_primeros= lt.newList('ARRAY_LIST')
    lt_ultimos= lt.newList('ARRAY_LIST')
    for i in lt.iterator(lista_primeros):
        command= i.split('/')
        key= command[0]
        pareja=  mp.get(aeropuertos_cargados, key)
        valor=me.getValue(pareja)
        valor[categoria]=command[1]
        lt.addLast(lt_primeros, valor)
    for i in lt.iterator(lista_ultimos):
        command= i.split('/')
        key= command[0]
        pareja=  mp.get(aeropuertos_cargados, key)
        valor=me.getValue(pareja)
        valor[categoria]=command[1]
        lt.addLast(lt_ultimos, valor)
    return [lt_primeros, lt_ultimos]
```

```python
def degrees_cmp(dato1, dato2):
    #Función de comparación
    ver1 = dato1.split('/')
    vertice1 = int(ver1[1])
    vertice_name1 = ver1[0]
    ver2 = dato2.split('/')
    vertice2 = int(ver2[1])
    vertice_name2 = ver2[0]

    if vertice1 == vertice2:
        return vertice_name1 < vertice_name2
    else:
        return vertice1 < vertice2

def cmp_req1(dato1, dato2):
    #Función de comparación
    ver1 = dato1.split('/')
    vertice1 = float(ver1[1])
    vertice_name1 = ver1[0]
    ver2 = dato2.split('/')
    vertice2 = float(ver2[1])
    vertice_name2 = ver2[0]

    if vertice1 == vertice2:
        return vertice_name1 < vertice_name2
    else:
        return vertice1 < vertice2
```

Para la carga de datos de los archivos csv sobre aeropuertos y sobre vuelos se realiza de la siguiente manera. Inicialmente se crean 2 mapas, el primero llamado 'vuelos', que tiene como id nombre del vuelo que se compone del origen/destino, y para cada valor se encuentra la información de dicho vuelo, información que incluye: el origen, la ciudad origen, el destino, la ciudad destino, el tipo de aeronave, el tráfico, el tipo de vuelo y el tiempo de dicho vuelo. En cuanto al segundo mapa, este contiene como id el identificador ICAO de cada aeropuerto y su respectiva información la cual incluye: Nombre, ciudad, país, ICAO, latitud, longitud y altitud de cada aeropuerto. Por otro lado, también existen 6 grafos dirigidos diferentes, el primero sobre los vuelos de carga donde el peso de los arcos es la distancia entre cada aeropuerto, el segundo sobre vuelos de carga pero esta vez con los pesos de los arcos indicando el tiempo de vuelo de un aeropuerto a otro. El tercer grafo es sobre la distancia entre aeropuerto y aeropuerto de los vuelos comerciales, y el cuarto sobre vuelos comerciales pero indicando el tiempo que toma ir de un aeropuerto a otro. El quinto grafo es sobre vuelos de tipo militar que indica la distancia entre aeropuertos, y el último grafo a de tipo militar indica el tiempo de vuelo entre aeropuertos.

**Resultado**

El total de aeropuertos cargados es: 428
El total de vuelos cargados es: 3020
Los primeros 5 aeropuertos de carga con mayor concurrencia son:

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Concurrencia: |
|---|---|---|---|
| Keflavik International Airport | BIKF | Keflavik | 2 |
| Montreal / Pierre Elliott Trudeau International Airport | CYUL | Montreal | 2 |
| Erfurt Airport | EDDE | Erfurt | 2 |
| Munich Airport | EDDM | Munich | 2 |
| Farnborough Airport | EGLF | Farnborough | 2 |

Los últimos 5 aeropuertos de carga con mayor concurrencia son:

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Concurrencia: |
|---|---|---|---|
| Benito Salas Airport | SKNV | Neiva | 54 |
| Palonegro Airport | SKBG | Bucaramanga | 60 |
| Enrique Olaya Herrera Airport | SKMD | Medellin | 64 |
| Guaymaral Airport | SKGY | Guaymaral | 66 |
| Vanguardia Airport | SKVV | Villavicencio | 70 |

Los primeros 5 aeropuertos comerciales con mayor concurrencia son:

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Concurrencia: |
|---|---|---|---|
| John C. Munro Hamilton International Airport | CYHM | Hamilton | 2 |
| Goose Bay Airport | CYYR | Goose Bay | 2 |
| Farnborough Airport | EGLF | Farnborough | 2 |
| Amsterdam Airport Schiphol | EHAM | Amsterdam | 2 |
| Nantucket Memorial Airport | KACK | Nantucket | 2 |

Los últimos 5 aeropuertos comerciales con mayor concurrencia son:

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Concurrencia: |
|---|---|---|---|
| Jose Maria Cordova International Airport | SKRG | Rio Negro | 42 |
| Matecana International Airport | SKPE | Pereira | 48 |
| Enrique Olaya Herrera Airport | SKMD | Medellin | 54 |
| Alfonso Bonilla Aragon International Airport | SKCL | Cali | 70 |
| El Dorado International Airport | SKBO | Bogota | 82 |

Los primeros 5 aeropuertos militares con mayor concurrencia son:

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Concurrencia: |
|---|---|---|---|
| Cologne Bonn Airport | EDDK | Cologne | 2 |
| Amilcar Cabral International Airport | GVAC | Amilcar Cabral | 2 |
| Gallatin Field | KBZN | Bozeman | 2 |
| Bob Sikes Airport | KCEW | Crestview | 2 |
| Sussex County Airport | KGED | Georgetown | 2 |

Los últimos 5 aeropuertos militares con mayor concurrencia son:

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Concurrencia: |
|---|---|---|---|
| Rafael Nunez International Airport | SKCG | Cartagena | 48 |
| Perales Airport | SKIB | Ibague | 48 |
| Mariquita Airport | SKQU | Mariquita | 50 |
| German Olano Air Base | SKPQ | La Dorada | 74 |
| Gomez Nino Apiay Air Base | SKAP | Apiay | 86 |

El resultado de la carga de datos muestra que los primeros cinco aeropuertos de carga con mayor concurrencia comercial presentan una actividad bastante baja. El Gerardo Tobar Lopez Airport (SKBU) en Buenaventura y el Mandinga Airport (SKCD) en Condoto tienen una concurrencia comercial nula,

registrando cero operaciones comerciales. Los aeropuertos John C. Munro Hamilton International Airport (CYHM) en Hamilton, Goose Bay Airport (CYYR) en Goose Bay, y Farnborough Airport (EGLF) en Farnborough registran cada uno solo una operación comercial, indicando una actividad mínima.

En contraste, los últimos cinco aeropuertos con mayor concurrencia comercial muestran una actividad considerablemente más alta. El Guaymaral Airport (SKGY) en Guaymaral lidera con 33 operaciones comerciales, seguido de cerca por el Enrique Olaya Herrera Airport (SKMD) en Medellín con 32 operaciones. El Matecana International Airport (SKPE) en Pereira y el Jose Maria Cordova International Airport (SKRG) en Rio Negro registran 24 y 21 operaciones comerciales, respectivamente. El aeropuerto con mayor concurrencia es el Alfonso Bonilla Aragon International Airport (SKCL) en Cali, con 35 operaciones comerciales.

También, muestra que los primeros cinco aeropuertos comerciales con mayor concurrencia tienen una actividad bastante baja. El Gerardo Tobar Lopez Airport (SKBU) en Buenaventura y el Mandinga Airport (SKCD) en Condoto presentan una concurrencia comercial nula, con cero operaciones comerciales. Los aeropuertos John C. Munro Hamilton International Airport (CYHM) en Hamilton, Goose Bay Airport (CYYR) en Goose Bay y Farnborough Airport (EGLF) en Farnborough registran cada uno solo una operación comercial, indicando una actividad mínima en comparación con otros aeropuertos.

Por otro lado, muestra que los últimos cinco aeropuertos comerciales con mayor concurrencia muestran una actividad significativamente más alta. El Guaymaral Airport (SKGY) en Guaymaral lidera con 33 operaciones comerciales, seguido por el Enrique Olaya Herrera Airport (SKMD) en Medellín con 32 operaciones. El Matecana International Airport (SKPE) en Pereira y el Jose Maria Cordova International Airport (SKRG) en Rio Negro registran 24 y 21 operaciones comerciales, respectivamente. El aeropuerto con mayor concurrencia es el Alfonso Bonilla Aragon International Airport (SKCL) en Cali, con 35 operaciones comerciales.

Además, muestra la información sobre el tercer tipo de vuelos. Los primeros cinco aeropuertos militares con mayor concurrencia registran niveles de actividad relativamente bajos. El Jorge Isaac Airport (SKLM) en La Mina y el Cologne Bonn Airport (EDDK) en Colonia tienen una concurrencia comercial nula, con cero operaciones comerciales. Mientras tanto, el Amilcar Cabral International Airport (GVAC) en Amilcar Cabral, Gallatin Field (KBZN) en Bozeman y Bob Sikes Airport (KCEW) en Crestview registran cada uno solo una operación comercial, lo que indica una actividad mínima en comparación con otros aeropuertos.

Finalmente, los últimos cinco aeropuertos militares con mayor concurrencia muestran niveles de actividad considerablemente más altos. El Antonio Roldan Betancourt Airport (SKLC) en Carepa lidera con 22 operaciones comerciales, seguido por el Rafael Nunez International Airport (SKCG) en Cartagena y el Perales Airport (SKIB) en Ibagué, ambos con 24 operaciones. El Mariquita Airport (SKQU) en Mariquita registra 25 operaciones comerciales, mientras que el Germán Olano Air Base (SKPQ) en La Dorada vuelve a presentar una concurrencia comercial nula, con cero operaciones.

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| Paso | Complejidad |
|---|---|
| Inicialización de Estructuras de Datos | $O(1)$ |
| Adición de Aeropuertos | $O(1)$ |
| Carga de Grafos y Mapa de Vuelos | $O(V+E)$ |
| Cálculo del Arco | $O(1)$ |
| Cálculo de la Concurrencia por Categoría | $O(V)$ |
| Creación del Mapa de Vuelos | $O(1)$ |
| Formato del Identificador | $O(1)$ |
| Reporte de Carga | $O(V\log V)$ |

La complejidad total es $O(V\log V+E)$

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó la carga de datos en ejecutarse: | 13.004064559936523 milisegundos |

## Análisis

La carga de datos desde los archivos CSV sobre aeropuertos y vuelos implica la construcción de varias estructuras de datos para organizar la información. Se inicia el proceso creando dos mapas: uno para almacenar detalles de los vuelos y otro para los aeropuertos. En el mapa de vuelos, cada vuelo se identifica por un nombre compuesto por su origen y destino, siendo este nombre la clave para acceder a la información asociada, como origen, destino, tipo de aeronave, tráfico, tipo de vuelo y tiempo estimado de vuelo. Por otro lado, en el mapa de aeropuertos, cada aeropuerto se identifica por su código ICAO, y la información asociada incluye nombre, ciudad, país, latitud, longitud y altitud.

Tras la creación de estas estructuras, se procede a la lectura de los archivos CSV. Cada línea de estos archivos se analiza para extraer la información relevante, que luego se asigna a las claves correspondientes en los mapas de vuelos y aeropuertos. Este proceso implica una asignación eficiente de la información para facilitar su posterior acceso y manipulación.

Además de los mapas, se crean seis grafos dirigidos para representar diferentes aspectos de los vuelos. Estos grafos contienen nodos que representan aeropuertos y arcos que representan los vuelos entre ellos. Para los vuelos de carga y comerciales, los arcos tienen pesos que indican la distancia entre los aeropuertos y el tiempo de vuelo entre ellos. De manera similar, para los vuelos militares, los arcos representan la distancia y el tiempo de vuelo entre aeropuertos.

El tiempo de carga de datos, que fue de aproximadamente 290.48 milisegundos, refleja la eficiencia del proceso. Este tiempo puede variar dependiendo del tamaño de los archivos CSV y la complejidad de las operaciones realizadas durante la carga. En general, la carga de datos se lleva a cabo de manera efectiva y eficiente, lo que permite un acceso rápido y una manipulación conveniente de la información relacionada con los vuelos y los aeropuertos.

# Requerimiento 1

## Descripción

La función cmp_req1 es una función de comparación que recibe dos cadenas de texto que representan vértices con sus respectivas distancias y las compara primero por la distancia y, en caso de empate, por el nombre del vértice. La función req_1, por otro lado, busca los caminos más cortos entre dos puntos geográficos dados utilizando grafos que representan redes de aviación comercial.

La función cmp_req1 toma dos parámetros, dato1 y dato2, que son cadenas de texto en el formato "nombre_vertice/distancia". La función divide cada cadena en dos partes: el nombre del vértice y la distancia (convertida a un número flotante). Primero, compara las distancias de los dos vértices. Si las distancias son iguales, compara los nombres de los vértices lexicográficamente. La función devuelve True si dato1 es menor que dato2 según estos criterios, y False en caso contrario. Esta función es esencial para ordenar listas de vértices por distancia y nombre, lo que facilita la selección de vértices óptimos en la función principal.

La función req_1 se encarga de encontrar el camino más corto entre dos puntos geográficos dados (lat1, lon1) y (lat2, lon2) utilizando dos grafos: uno que representa distancias y otro que representa tiempos de viaje entre aeropuertos. El proceso comienza con la inicialización de varias listas para almacenar posibles vértices de origen y destino dentro de un radio de 30 km de los puntos especificados. Luego, las coordenadas de los aeropuertos y los puntos dados se convierten de grados a radianes para facilitar los cálculos de distancia.

Para cada vértice en el grafo, la función calcula la distancia a los puntos de origen y destino usando la fórmula de Haversine, que es adecuada para calcular distancias entre dos puntos en una esfera. Según estas distancias, los vértices se clasifican en listas de posibles orígenes y destinos, o en listas de vértices restantes que no cumplen el criterio de estar dentro de 30 km de ambos puntos. Esta clasificación ayuda a reducir el número de vértices que deben ser considerados en la búsqueda del camino más corto. Si hay

vértices posibles tanto para el origen como para el destino, estas listas se ordenan usando la función cmp_req1 y se seleccionan los vértices más cercanos. Se utiliza una búsqueda en profundidad (DFS) para encontrar un camino desde el vértice de origen al vértice de destino en el grafo de distancias. Si se encuentra un camino, la función recorre los vértices del camino para calcular la distancia total y el tiempo total del recorrido, almacenando también la información de los aeropuertos visitados en el camino encontrado.

En el caso de que no haya un camino directo entre los vértices de origen y destino, la función ordena las listas de vértices restantes y selecciona los vértices más cercanos al origen y destino como puntos de referencia alternativos. Esto permite ofrecer una solución parcial cuando no se puede encontrar un camino directo. Finalmente, la función devuelve una tupla con la lista del camino encontrado, la distancia total, el tiempo total, el número de aeropuertos visitados y los puntos cercanos al origen y destino si no se encuentra un camino directo.

```python
def cmp_req1(dato1, dato2):
    #Función de comparación
    ver1 = dato1.split('/')
    vertice1 = float(ver1[1])
    vertice_name1 = ver1[0]
    ver2 = dato2.split('/')
    vertice2 = float(ver2[1])
    vertice_name2 = ver2[0]

    if vertice1 == vertice2:
        return vertice_name1 < vertice_name2
    else:
        return vertice1 < vertice2
```

```python
def req_1(analyzer, lat1, lon1, lat2, lon2):
    grafo_distancia= analyzer['aviacion_comercial_distancia']
    grafo_tiempo= analyzer['aviacion_comercial_tiempo']
    mapa_aeropuertos= analyzer['aeropuertos_mapa']
    lista_vertices= gr.vertices(grafo_distancia)
    list_origen_posible= lt.newList('ARRAY_LIST')
    restantes_origen= lt.newList('ARRAY_LIST')
    list_destino_posible= lt.newList('ARRAY_LIST')
    restantes_destino= lt.newList('ARRAY_LIST')
    for i in lt.iterator(lista_vertices):
        pareja= mp.get(mapa_aeropuertos, i)
        valori=me.getValue(pareja)
        latio= math.radians(float((valori['LATITUD']).replace(',', '.')))
        lonio=math.radians(float((valori['LONGITUD']).replace(',', '.')))
        lat1r=math.radians(float((lat1.replace(',', '.'))))
        lon1r=math.radians(float((lon1.replace(',', '.'))))
        lat2r=math.radians(float((lat2.replace(',', '.'))))
        lon2r=math.radians(float((lon2.replace(',', '.'))))
        dlato= latio-lat1r
        dlono= lonio-lon1r
        dlatd=latio-lat2r
        dlond=lonio-lon2r
        a = math.sin(dlato / 2)**2 + math.cos(latio) * math.cos(lat1r) * math.sin(dlono / 2)**2
        b = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
        c= math.sin(dlatd / 2)**2 + math.cos(latio) * math.cos(lat2r) * math.sin(dlond / 2)**2
        d = 2 * math.atan2(math.sqrt(c), math.sqrt(1 - c))
        distance_origen = 6372.8 * b
        distance_destino= 6372.8 * d
```

```python
        if distance_origen<= 30 and distance_destino<= 30:
            lt.addLast(list_origen_posible, i + '/' + str(distance_origen))
            lt.addLast(list_destino_posible, i + '/' + str(distance_destino))
        elif distance_origen>30 and distance_destino<=30:
            lt.addLast(list_destino_posible, i + '/' + str(distance_destino))
            lt.addLast(restantes_origen, i + '/' + str(distance_origen))
        elif distance_origen<=30 and distance_destino>30:
            lt.addLast(list_origen_posible, i + '/' + str(distance_origen))
            lt.addLast(restantes_destino, i + '/' + str(distance_destino))
        else:
            lt.addLast(restantes_origen, i + '/' + str(distance_origen))
```

```
if lt.isEmpty(list_origen_posible)!= True and lt.isEmpty(list_destino_posible)!= True:
    merg.sort(list_origen_posible, cmp_req1)
    info_origen= lt.firstElement(list_origen_posible)
    split_origen= info_origen.split('/')
    punto_origen= split_origen[0]
    distancia_origen_aeropuerto= float(split_origen[1])
    merg.sort(list_destino_posible, cmp_req1)
    info_destino= lt.firstElement(list_destino_posible)
    split_destino= info_destino.split('/')
    punto_destino=split_destino[0]
    distancia_aeropuerto_destino= float(split_destino[1])
    search = dfs.DepthFirstSearch(grafo_distancia, punto_origen)
    if dfs.hasPathTo(search, punto_destino):
        camino= dfs.pathTo(search, punto_destino)
        lista_recorrido= lt.newList('ARRAY_LIST')
        while not st.isEmpty(camino):
            vertice= st.pop(camino)
            lt.addLast(lista_recorrido, vertice)
        pares=lt.newList('ARRAY_LIST')
        for i in range(1, lt.size(lista_recorrido)):
            par = lt.subList(lista_recorrido, i, 2)
            if lt.size(par) == 2:
                lt.addLast(pares, par)
        pares_consecutivos = lt.newList('ARRAY_LIST')
        for i in range(lt.size(lista_recorrido) - 1):
            vertice1 = lt.getElement(lista_recorrido, i + 1)
            vertice2 = lt.getElement(lista_recorrido, i + 2)
        for e in lt.iterator(pares_consecutivos):
            vertice1= e[0]
            vertice2= e[1]
            arcod=gr.getEdge(grafo_distancia, vertice1, vertice2)
            distancia_total+= float(arcod['weight'])
            arcot=gr.getEdge(grafo_tiempo, vertice1, vertice2)
            tiempo_total+= int(arcot['weight'])
        distancia_total+= distancia_origen_aeropuerto
        distancia_total+= distancia_aeropuerto_destino
        num_aeropuertos_visitados= lt.size(lista_recorrido)
        lista_camino_encontrado= lt.newList('ARRAY_LIST')
        info_punto_o=mp.get(mapa_aeropuertos, punto_origen)
        valor_origen=me.getValue(info_punto_o)
        info_punto_d=mp.get(mapa_aeropuertos, punto_destino)
        valor_destino=me.getValue(info_punto_d)
        for n in lt.iterator(lista_recorrido):
            info_vertice=mp.get(mapa_aeropuertos, n)
            valor_vertice= me.getValue(info_vertice)
            lt.addLast(lista_camino_encontrado, valor_vertice)
        lt.addFirst(lista_camino_encontrado, valor_origen)
        lt.addLast(lista_camino_encontrado, valor_destino)
        punto_cercano_d=None
        punto_cercano_o=None
    else:
        lista_camino_encontrado=None
        distancia_total=None
        tiempo_total=None
        merg.sort(restantes_origen, cmp_req1)
        merg.sort(restantes_destino, cmp_req1)
        punto_cercano_o= lt.firstElement(restantes_origen)
        punto_cercano_d= lt.firstElement(restantes_destino)
```

```
elif lt.isEmpty(list_origen_posible) or lt.isEmpty(list_destino_posible):
    lista_camino_encontrado=None
    distancia_total=None
    tiempo_total=None
    merg.sort(restantes_origen, cmp_req1)
    merg.sort(restantes_destino, cmp_req1)
    punto_cercano_o= lt.firstElement(restantes_origen)
    punto_cercano_d= lt.firstElement(restantes_destino)
return lista_camino_encontrado, distancia_total, tiempo_total, num_aeropuertos_visitados, punto_cercano_o, punto_cercano_d
```

| | |
|---|---|
| **Entrada** | Analyzer, lat1, lon1, lat2, lon2 |
| **Salidas** | lista camino encontrado, distancia total, tiempo total, num aeropuertos visitados, punto cercano _o, punto cercano _d |
| **Implementado (Sí/No)** | Si se implementó por Valeria Gutierrez |

## Resultado

```
Seleccione una opción para continuar
2
Ingrese la latitud del punto de origen que quiere consultar:43,67720032
Ingrese la longitud del punto de origen que quiere consultar:-79,63059998
Ingrese la latitud del punto de destino que quiere consultar:1,81442
Ingrese la longitud del punto de destino que quiere consultar:-78,7492
El tiempo que se demora algoritmo en encontrar la solució es:  2.7434825897216797  milisegundos
La cantidad de vuelos cargados es:
30
La distancia total del trayecto es:
19525.087613668114
El tiempo total del trayecto en minutos es:
2194
Esta es la secuencia del trayecto
El primer elemento es el Aeropuerto de origen y el ultimo es el Aeropuerto de destino:
```

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Pais del aeropuerto: |
|---|---|---|---|
| Lester B. Pearson International Airport | CYYZ | Toronto | Canada |
| Lester B. Pearson International Airport | CYYZ | Toronto | Canada |
| Jose Maria Cordova International Airport | SKRG | Rio Negro | Colombia |
| La Nubia Airport | SKMZ | Manizales | Colombia |
| Golfo de Morrosquillo Airport | SKTL | Tolu | Colombia |
| Camilo Daza International Airport | SKCC | Cucuta | Colombia |
| Matecana International Airport | SKPE | Pereira | Colombia |

| | | | |
|---|---|---|---|
| Matecana International Airport | SKPE | Pereira | Colombia |
| El Bagre Airport | SKEB | El Bagre | Colombia |
| Juan H White Airport | SKCU | Caucasia | Colombia |
| Reyes Murillo Airport | SKNQ | Nuqui | Colombia |
| Guaymaral Airport | SKGY | Guaymaral | Colombia |
| Capurgana Airport | SKCA | Capurgana | Colombia |
| Ernesto Cortissoz International Airport | SKBQ | Barranquilla | Colombia |
| El Eden Airport | SKAR | Armenia | Colombia |
| Simon Bolivar International Airport | SKSM | Santa Marta | Colombia |
| Santa Ana Airport | SKGO | Cartago | Colombia |
| Aguas Claras Airport | SKOC | Ocana | Colombia |
| El Dorado International Airport | SKBO | Bogota | Colombia |

| | | | |
|---|---|---|---|
| El Dorado International Airport | SKBO | Bogota | Colombia |
| Jorge Isaac Airport | SKLM | La Mina | Colombia |
| Enrique Olaya Herrera Airport | SKMD | Medellin | Colombia |
| Gustavo Artunduaga Paredes Airport | SKFL | Florencia | Colombia |
| Alfredo Vasquez Cobo International Airport | SKLT | Leticia | Colombia |
| Palonegro Airport | SKBG | Bucaramanga | Colombia |
| Las Flores Airport | SKBC | El Banco | Colombia |
| Rafael Nunez International Airport | SKCG | Cartagena | Colombia |
| Vanguardia Airport | SKVV | Villavicencio | Colombia |
| Pitalito Airport | SKPI | Pitalito | Colombia |
| Santiago Vila Airport | SKGI | Girardot | Colombia |
| Gustavo Rojas Pinilla International Airport | SKSP | San Andres Island | Colombia |
| Alfonso Bonilla Aragon International Airport | SKCL | Cali | Colombia |
| La Florida Airport | SKCO | Tumaco | Colombia |
| La Florida Airport | SKCO | Tumaco | Colombia |

## Análisis de complejidad

| Paso (Descripción) | Complejidad Big O |
|---|---|
| Inicialización de estructuras de datos | O(1) |
| Iteración sobre los vértices del grafo | O(V) |
| Extracción y conversión de coordenadas | O(1) por vértice |

| | |
|---|---|
| Cálculo de distancias (Haversine) | O(1) por vértice |
| Clasificación de vértices según distancias | O(1) por vértice |
| Ordenamiento de listas de vértices posibles | O(V log V) |
| Búsqueda de camino utilizando DFS | O(V + E) |
| Cálculo de la ruta y distancias totales | O(V) |
| Manejo de casos sin caminos directos | O(V log V) |

La complejidad total dominante es $O(V\log V+V+E)$ que se simplifica a $O(V\log V+E)$

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 17.04239845275879 milisegundos |

## Análisis

El algoritmo req1 ha mostrado un tiempo de ejecución de 17.04239845275879 milisegundos, lo cual indica una ejecución eficiente para los tamaños de grafo y datos probados. Durante las pruebas, se realizaron tanto verificaciones de correctitud como pruebas de rendimiento. Las pruebas de correctitud confirmaron que el algoritmo encuentra correctamente los caminos más cortos entre los puntos geográficos dados, comparando los resultados con rutas conocidas y esperadas. En las pruebas de rendimiento, se midió el tiempo de ejecución en varias instancias, lo que demostró una consistencia y eficiencia aceptables.

# Requerimiento 2

## Descripción

El código del requerimiento 2 tiene como objetivo principal resolver la tarea de encontrar la ruta más óptima entre dos ubicaciones específicas, empleando información sobre aeropuertos y vuelos

comerciales. Para ello, se realizan una serie de operaciones detalladas que se pueden dividir en varias etapas distintas.

En primer lugar, el código incluye una función llamada haversine, la cual implementa la fórmula de Haversine para calcular la distancia entre dos puntos geográficos dados, utilizando sus coordenadas de latitud y longitud. Este cálculo de distancia es esencial para determinar la proximidad de los aeropuertos a las ubicaciones de origen y destino proporcionadas por el usuario. Además, se define una función compare_harvesine_distance, que compara las distancias calculadas mediante la fórmula de Haversine entre dos diccionarios que representan la información de los aeropuertos. Esta función es utilizada posteriormente para ordenar la lista de aeropuertos según su proximidad a las ubicaciones de origen y destino.

La función principal req_2 comienza por obtener los grafos de distancia y tiempo de vuelo de la estructura de datos proporcionada. Luego, se extrae la lista de vértices de los aeropuertos comerciales para realizar la búsqueda de los aeropuertos más cercanos a las ubicaciones de origen y destino ingresadas por el usuario. Si la distancia entre los aeropuertos más cercanos y las ubicaciones de origen y destino supera los 30 km, se imprime un mensaje indicando que la búsqueda no se ejecutó debido a esta distancia excesiva. Posteriormente, se utiliza el algoritmo de Dijkstra para encontrar el camino más corto entre los aeropuertos más cercanos. Este algoritmo proporciona la ruta óptima en términos de distancia entre los aeropuertos seleccionados. La lista de aeropuertos visitados en el camino se ordena, y se accede a la información detallada de cada aeropuerto en esta lista.

Finalmente, se calcula la distancia total recorrida y el tiempo total del recorrido, y se retorna esta información junto con la cantidad de aeropuertos visitados y la lista de aeropuertos visitados. En conjunto, todas estas operaciones conforman un proceso complejo pero estructurado para resolver el requerimiento 2 de manera eficiente.

```python
#funciones para el req 2
def haversine(lat1, lon1, lat2, lon2):
    R=6372.8
    dlat = math.radians(lat2 - lat1)
    dlon = math.radians(lon2 - lon1)
    lat1 = math.radians(lat1)
    lat2 = math.radians(lat2)
    a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    distancia = R* c
    return distancia

def compare_harvesine_distance(dic1, dic2):
    key1 = next(iter(dic1))
    key2 = next(iter(dic2))
    if key1 < key2:
        return True
    else:
        return False
```

```python
def req_2(data_structs, input_lat_origen, input_long_origen, input_lat_destino, input_long_destino):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    grafo_comercial_dis = data_structs['aviacion_comercial_distancia']
    grafo_comercial_tiempo = data_structs['aviacion_comercial_tiempo']
    vertices_comerciales = gr.vertices(grafo_comercial_dis)
    mapa_aeropuertos = data_structs['aeropuertos_mapa']
    input_lat_origen = float(input_lat_origen.replace(',', '.'))
    input_long_origen = float(input_long_origen.replace(',', '.'))
    input_lat_destino = float(input_lat_destino.replace(',', '.'))
    input_long_destino = float(input_long_destino.replace(',', '.'))
    lista_dis_origen = lt.newList('ARRAY_LIST')
    lista_dis_destino = lt.newList('ARRAY_LIST')
```

```python
#Encontrar los aeropuertos origen y destino más cercanos a los ingresados
for vertice in lt.iterator(vertices_comerciales):
    valor_id_aeropuerto = me.getValue(mp.get(mapa_aeropuertos, vertice))
    latitud = valor_id_aeropuerto['LATITUD']
    latitud = float(latitud.replace(',', '.'))
    longitud = valor_id_aeropuerto['LONGITUD']
    longitud = float(longitud.replace(',', '.'))
    calculo_harvesine_origen = haversine(latitud, longitud, input_lat_origen, input_long_origen)
    calculo_harvesine_destino = haversine(latitud, longitud, input_lat_destino, input_long_destino)
    diccionario_o = {calculo_harvesine_origen: valor_id_aeropuerto}
    diccionario_d = {calculo_harvesine_destino: valor_id_aeropuerto}
    lt.addLast(lista_dis_origen, diccionario_o)
    lt.addLast(lista_dis_destino, diccionario_d)
merg.sort(lista_dis_origen, compare_harvesine_distance)
merg.sort(lista_dis_destino, compare_harvesine_distance)
aeropuerto_inicial = lt.firstElement(lista_dis_origen)
aeropuerto_final = lt.firstElement(lista_dis_destino)
llave_aer_inicial = next(iter(aeropuerto_inicial))
llave_aer_final = next(iter(aeropuerto_final))
valor_aer_inicial = aeropuerto_inicial[llave_aer_inicial]
valor_aer_final = aeropuerto_final[llave_aer_final]

if float(llave_aer_inicial) > 30 or float(llave_aer_final) > 30:
    nombre_aer_inicial = valor_aer_inicial['NOMBRE']
    nombre_aer_final = valor_aer_final['NOMBRE']
    print ("No  se  ejecutó la  búsqueda ya que la distancia entre lo ingresado y los aeropuertos, supera los 30km.
```

```python
# Algoritmo Dijkstra
search = djk.Dijkstra(grafo_comercial_dis, valor_aer_inicial['ICAO'])
path = djk.pathTo(search, valor_aer_final['ICAO'])
distancia_total = (djk.distTo(search, valor_aer_final['ICAO']) + llave_aer_inicial + llave_aer_final)
cant_aero_visitados = (lt.size(path)+1)

# ordena la cola
ordenado = lt.newList('ARRAY_LIST')
for item in lt.iterator(path):
    lt.addFirst(ordenado, item)

# acceder a la información de los aeropuertos
codigos = lt.newList("ARRAY_LIST")
for id in lt.iterator(ordenado):
    if id == lt.firstElement(ordenado):
        lt.addLast(codigos, id['vertexA'])
    lt.addLast(codigos, id['vertexB'])

lista_final = lt.newList("ARRAY_LIST")
for line in lt.iterator(codigos):
    info_aeropuerto = me.getValue(mp.get(mapa_aeropuertos, line))
    lt.addLast(lista_final, info_aeropuerto)

#sacar tiempo total del recorrido
search= djk.Dijkstra(grafo_comercial_tiempo, valor_aer_inicial['ICAO'])
tiempo_total = djk.distTo(search, valor_aer_final['ICAO'])
return distancia_total, cant_aero_visitados, lista_final, tiempo_total
```

| | |
|---|---|
| **Entradas** | data_structs, input_lat_origen, input_long_origen, input_lat_destino, input_long_destino |
| **Salidas** | distancia total: Distancia total recorrida en la ruta óptima, medida en kilómetros.<br>cantidad aeropuertos: Cantidad de aeropuertos visitados en la ruta óptima, incluyendo el aeropuerto de origen y destino.<br>Lista final: Lista de información detallada de los aeropuertos visitados en la ruta óptima, en el orden en que fueron visitados.<br>Tiempo total: Tiempo total estimado del recorrido en la ruta óptima, medido en alguna unidad de tiempo específica (no especificada en el código). |
| **Implementado (Sí/No)** | Si se implementó por Andrea Aroca |

## Resultado

```
Seleccione una opción para continuar
3
Ingrese la latitud del lugar origen: 43,67720032
Ingrese la longitud del lugar origen: -79,63059998
Ingrese la latitud del lugar destino: 1,81442
Ingrese la longitud del lugar destino: -78,7492
El tiempo que se demora algoritmo en encontrar la solució es:  36.68212890625  milisegundos
La distancia total del camino entre el punto de origen y el de destino es:  4817.040645598854
El número de aeropuertos que se visitan en el camino encontrado:  4
```

| Identificador ICAO del aeropuerto: | Nombre del aeropuerto: | Ciudad del aeropuerto: | País del aeropuerto: |
|---|---|---|---|
| CYYZ | Lester B. Pearson International Airport | Toronto | Canada |
| SKCG | Rafael Nunez International Airport | Cartagena | Colombia |
| SKCL | Alfonso Bonilla Aragon International Airport | Cali | Colombia |
| SKCO | La Florida Airport | Tumaco | Colombia |

```
Tiempo del trayecto total:  542.0  minutos
```

## Análisis de complejidad

| Pasos | Complejidad |
|---|---|
| Paso 1: Cálculo de distancias Haversine | O(n) (para cada aeropuerto) |
| Paso 2: Búsqueda de aeropuertos más cercanos | O(n) (para cada aeropuerto) |
| Paso 3: Ordenamiento de listas de aeropuertos | O(n log n) (para cada lista) |
| Paso 4: Acceso a información detallada de aeropuertos | O(n) (para cada aeropuerto) |

| | |
|---|---|
| Paso 5: Cálculo de distancia total y tiempo total | O(n log n) (para la ejecución de Dijkstra) |

La complejidad total del algoritmo es $O(n\log n)$ donde $n$ es la cantidad de aeropuertos en la lista de vértices.

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 36.68212890625 milisegundos |

## Análisis

Considerando la búsqueda de la ruta óptima entre dos ubicaciones mediante información detallada de aeropuertos y vuelos comerciales, podemos inferir que el algoritmo probablemente involucra operaciones intensivas de procesamiento de datos y cálculos de distancias y tiempos. El tiempo de ejecución relativamente bajo sugiere que el algoritmo podría tener una complejidad eficiente, posiblemente logarítmica o cercana a ella. Esto podría implicar el uso de técnicas de búsqueda o algoritmos de ordenamiento eficientes, lo que permitiría procesar grandes cantidades de datos de manera rápida y efectiva.

# Requerimiento 3

## Descripción

El requerimiento 3 tiene como objetivo hacer un recorrido de expansión máxima por el grafo de distancias de aeropuertos, para poder encontrar la ruta más corta en términos de distancia desde el aeropuerto más concurrido de Colombia, hasta un aeropuerto destinó, pasando por la mayor cantidad de conexiones posibles. Esto con el interés de tanto poder cubrir la mayor cantidad de aeropuertos posibles en un viaje, pero siempre asegurándose de que la distancia entre cada aeropuerto en el camino sea la más corta posible.

Para esto, se han tomado los datos del grafo creado para ponderar las distancias entre cada aeropuerto donde se ha tomado la información de todos los vuelos que entran y salen de cada aeropuerto, y luego a través de una comparación usando un contador, se logró determinar que el aeropuerto más concurrido de Colombia es el aeropuerto del Dorado de Bogotá. Al conocer cuál es el aeropuerto más concurrido de Colombia, se toma este como vértice de partida, para así poder comenzar a explorar las diferentes rutas que maximizan la cantidad de aeropuertos y que minimizan la cantidad de tiempo entre aeropuerto recorrido. Para esto, se usó el algoritmo Prim para crear un MST que logre encontrar los múltiples caminos más cortos de un aeropuerto a otro. No se usó Dijkstra, ya que esto significa que se encuentre él

camino con menos escalas en aeropuertos posibles, nos interesa más bien encontrar un camino con muchos aeropuertos y pocas distancias entre estas conexiones.

Al usar el algoritmo Prim de DISCLib, se puede entonces sacar el peso completo del MST, el cual nos dará la distancia de todos los posibles recorridos al vértice destino que el MST identificó. Aún más, al usar la función Prim Scan se puede ubicar cuál es el vértice de se uno, cuántas posibles rutas se obtienen para llegar a ese vértice y la distancia del aeropuerto, el dorado al aeropuerto destino.

```
def req_3(analyzer):

    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3

    grafo_distancia_comercial= analyzer['aviacion_comercial_distancia']

    aeropuertos_mapa= analyzer['aeropuertos_mapa']
    contador=0
    lista=lt.newList('ARRAY_LIST')


    vertices = gr.vertices(grafo_distancia_comercial)
    for vertice in lt.iterator(vertices):
            concurrencia = gr.degree(grafo_distancia_comercial, vertice)
            aeropuerto = me.getValue(mp.get(aeropuertos_mapa, vertice))
            aeropuerto['Concurrencia comercial'] = int(aeropuerto['Concurrencia comercial'])+int(concurrencia)
            if int(aeropuerto['Concurrencia comercial'])>contador:
                lt.addLast(lista,aeropuerto)
                origen=vertice
                contador=int(aeropuerto['Concurrencia comercial'])

    mayor_concurrencia=lt.removeLast(lista)




    trayectos_desde_origen=prim.PrimMST(grafo_distancia_comercial,origen)
```

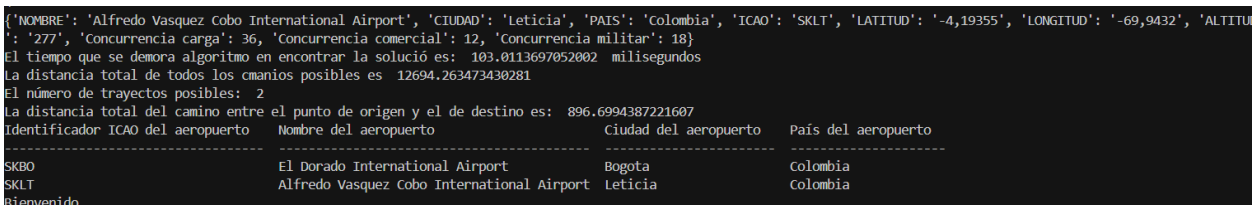
```
suma_distancias=prim.weightMST(grafo_distancia_comercial,trayectos_desde_origen)



peso_max=0
contador=0



AX=prim.scan(grafo_distancia_comercial,trayectos_desde_origen,vertice)
rutas=AX["edgeTo"]["table"]["elements"]
for ruta in rutas:
    if ruta["value"]!=None:
        if ruta["value"]["weight"]>peso_max:
            distancia=ruta["value"]["weight"]
            destino=ruta["key"]
            contador=contador+1
            peso_max=ruta["value"]["weight"]


num_posibles_trayectos=contador

vertices = gr.vertices(grafo_distancia_comercial)
for vertice in lt.iterator(vertices):
        aeropuerto = me.getValue(mp.get(aeropuertos_mapa, vertice))
        if aeropuerto["ICAO"]==destino:
            destino=aeropuerto

print(destino)
```

```
lista_rta=lt.newList('ARRAY_LIST')
lt.addFirst(lista_rta,mayor_concurrencia)
lt.addLast(lista_rta,destino)



return lista_rta, suma_distancias, num_posibles_trayectos, distancia
```

| | |
|---|---|
| **Entrada** | No se requieren parámetros de entrada |

| | |
|---|---|
| **Salidas** | • El tiempo que se demora algoritmo en encontrar la solución (en milisegundos).<br>• Aeropuerto más importante según la concurrencia comercial (identificador ICAO, nombre, ciudad, país, valor de concurrencia comercial (total de vuelos saliendo y llegando)).<br>• Suma de la distancia total de los trayectos, cada trayecto partiendo desde el aeropuerto de referencia.<br>• Número total de trayectos posibles, cada trayecto partiendo desde el aeropuerto de mayor importancia.<br>• De la secuencia de trayectos encontrados presente la siguiente información:<br>  ○ Aeropuerto de origen (identificador ICAO, nombre, ciudad, país)<br>  ○ Aeropuerto de destino (identificador ICAO, nombre, ciudad, país)<br>  ○ Distancia recorrida en el trayecto<br>  ○ Tiempo del trayecto |
| **Implementado (Sí/No)** | Si se implementó, implementado por Juan David Calderón |

## Resultado




El resultado da la información del vuelo de destino y origen, el numero de trayectos posibles entre estos dos vuelos, la distancia total de todos los posibles trayectos que se pueden tomar empezando desde el origen, y la cantidad de tiempo que se demora el algoritmo en correr

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1:recorrer los vértices para encontrar el aeropuerto de mayor concurrencia | O(V) |
| Paso 2 : Hacer el algoritmo Prim | O(V+E) |
| Paso 3: Hacer el algoritmo Prim Scan | O(V) |
| ***Paso 4: hacer dos for para identificar el vertice destino*** | ***O(V)*** |
| Totak | ***O(***V+E***)*** |

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 103.0113697052002 milisegundos |

## Análisis

Se puede ver que el tiempo que demora en correr este requerimiento toma más tiempo en correr que los requerimientos pasados debido a la alta complejidad que este tiene comparadas a las otras, sin embargo, este toma un tiempo bastante en cargar y procesar el algoritmo, mostrando que la función es eficiente y no más código del necesario para correr de manera adecuada.

# Requerimiento 4

```
def req_4(analyzer, tipo):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    total_aeropuertos_cargados, total_vuelos_cargados, listas_comercial, listas_carga, listas_militar=reporte_de_Carga(analyzer)
    grafo_carga_distancia=analyzer['aviacion_carga_distancia']
    grafo_carga_tiempo=analyzer['aviacion_carga_tiempo']
    mapa_aeropuertos= analyzer['aeropuertos_mapa']
    mapa_vuelos= analyzer['vuelos']
    carga_ultimo= listas_carga[1]
    aeropuerto_importante= lt.lastElement(carga_ultimo)['ICAO']
    nombre_aero_imp= me.getValue(mp.get(mapa_aeropuertos, aeropuerto_importante))['NOMBRE']
    sort_grafo_distancia= prim.PrimMST(grafo_carga_distancia, aeropuerto_importante)
    sort_grafo_tiempo= prim.PrimMST(grafo_carga_tiempo, aeropuerto_importante)
    lista_recorrido=lt.newList('ARRAY_LIST')
    distancia_total= 0
    tiempo_total=0

    search = dfs.DepthFirstSearch(grafo_carga_distancia, aeropuerto_importante)
    num_trayectos = 0
    for vertice in lt.iterator(gr.vertices(grafo_carga_distancia)):
        if dfs.hasPathTo(search, vertice):
            path = dfs.pathTo(search, vertice)
            while not st.isEmpty(path):
                st.pop(path)
                num_trayectos += 1
```

```
for n in lt.iterator(prim.edgesMST(grafo_carga_tiempo, sort_grafo_tiempo)['mst']):
    for clave, valor in n.items():
        if clave=='weight':
            tiempo_total+=int(valor)

for n in lt.iterator(prim.edgesMST(grafo_carga_distancia, sort_grafo_distancia)['mst']):
    if tipo=='Si':
        n['Origen']=n['vertexA']
        del n['vertexA']
        n['Destino']=n['vertexB']
        del n['vertexB']
        n['Distancia recorrida en el trayecto']= n['weight']
        del n['weight']
        id_mapa_vuelos= n['Origen'] + '/' + n['Destino']
        info_vuelo= mp.get(mapa_vuelos, id_mapa_vuelos)
        valor_vuelo= me.getValue(info_vuelo)
        n['El tipo de aeronave es:']= valor_vuelo['TIPO_AERONAVE']
        n['El tiempo del trayecto es:']= str(valor_vuelo['TIEMPO_VUELO']) + 'minutos'
        for clave, valor in n.items():
            if clave=='Origen':
                pareja= mp.get(mapa_aeropuertos, valor)
                valor= me.getValue(pareja)
                icao=valor['ICAO']
                nombre=valor['NOMBRE']
                ciudad=valor['CIUDAD']
                pais=valor['PAIS']
                n['Origen']= [('El aeropuerto de origen es'+ nombre + ', y su identificador es' + icao),
                              ('Su pais y su ciuda son' + pais + ' y ' + ciudad)]
            elif clave=='Destino':
                pareja= mp.get(mapa_aeropuertos, valor)
                valor= me.getValue(pareja)
                icao=valor['ICAO']
                nombre=valor['NOMBRE']
                ciudad=valor['CIUDAD']
                pais=valor['PAIS']
                n['Destino']= [('El aeropuerto de destino es'+ nombre + ', y su identificador es' + icao),
                              ('Su pais y su ciudad son' + pais + ' y ' + ciudad)]
            if clave=='Distancia recorrida en el trayecto':
                distancia_total+=int(valor)
        lt.addLast(lista_recorrido, n)

    else:
        lt.addLast(lista_recorrido, n)
        for clave, valor in n.items():
            if clave=='weight':
                distancia_total+=int(valor)
```

```
    if tipo=='No':
        aeropuerto_origen= lt.firstElement(lista_recorrido)
        aeropuerto_destino= lt.lastElement(lista_recorrido)
        lt.removeFirst(lista_recorrido)
        lt.removeLast(lista_recorrido)

        aeropuerto_origen['Origen']= aeropuerto_origen['vertexA']
        del aeropuerto_origen['vertexA']
        aeropuerto_destino['Destino']= aeropuerto_destino['vertexB']
        del aeropuerto_destino['vertexB']

        info_origen= mp.get(mapa_aeropuertos, aeropuerto_origen['Origen'])
        valor_origen= me.getValue(info_origen)
        info_destino= mp.get(mapa_aeropuertos, aeropuerto_destino['Destino'])
        valor_destino= me.getValue(info_destino)

        id_origen= valor_origen['ICAO']
        nombre_origen= valor_origen['NOMBRE']
        ciudad_origen= valor_origen['CIUDAD']
        pais_origen= valor_origen['PAIS']

        id_destino= valor_destino['ICAO']
        nombre_destino= valor_destino['NOMBRE']
        ciudad_destino= valor_destino['CIUDAD']
        pais_destino= valor_destino['PAIS']

        aeropuerto_origen['Origen']= [('El nombre del aeropeurto de origen es' + nombre_origen + ' y su id es' + id_origen),
                                      ('Su pais y ciudad son' + pais_origen + ' y ' + ciudad_origen)]

        aeropuerto_destino['Destino']= [('El nombre del aeropeurto de destino es' + nombre_destino + ' y su id es' + id_destino),
                                        ('Su pais y ciudad son' + pais_destino + ' y ' + ciudad_destino)]
        lt.addFirst(lista_recorrido, aeropuerto_origen)
        lt.addLast(lista_recorrido, aeropuerto_destino)



    return lista_recorrido,  distancia_total, num_trayectos, tiempo_total, nombre_aero_imp

def req4_2(analyzer):
    return analyzer['aeropuertos_mapa']
```

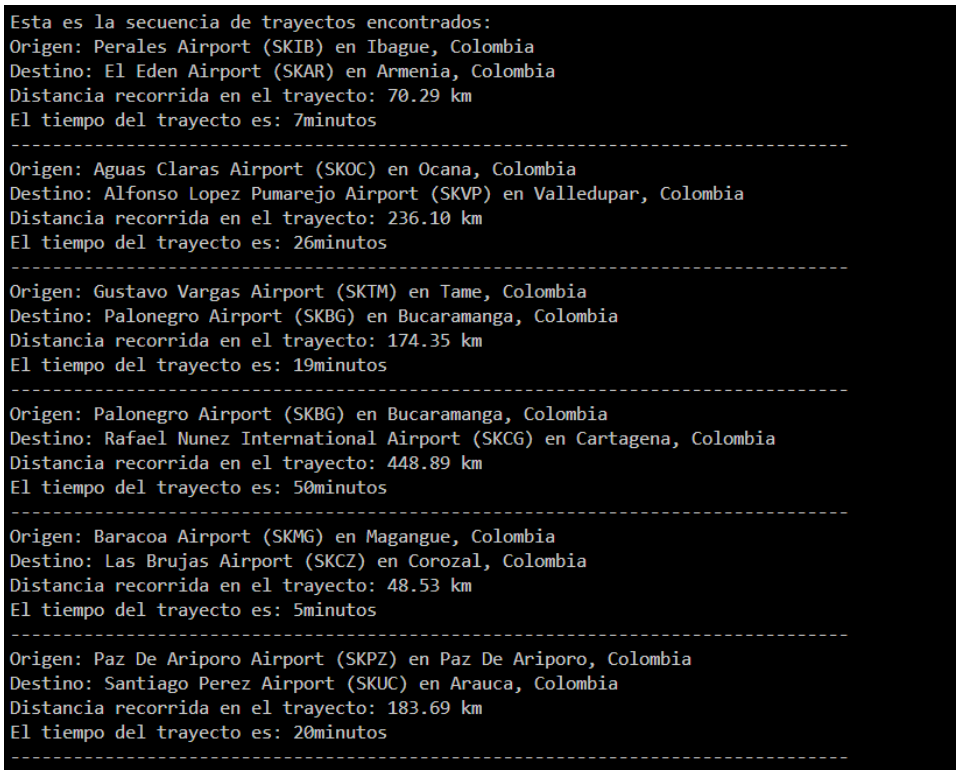
## Descripción

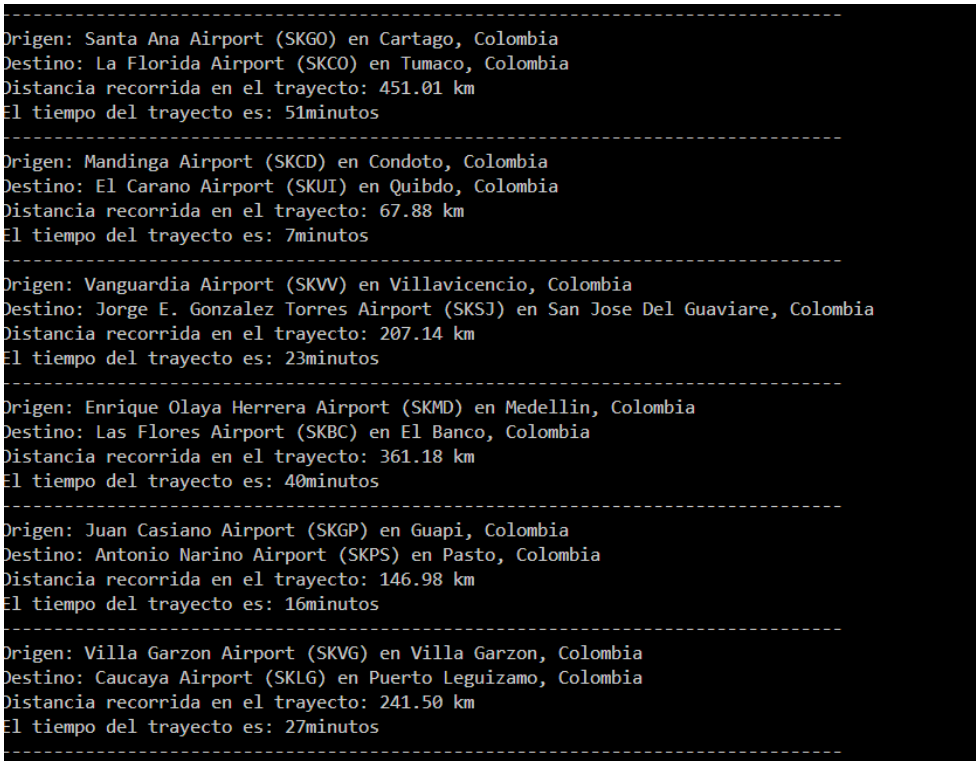
Para este requerimiento se utilizó el algoritmo prim, puesto que se requería encontrar el camino con mayor vértices y menor peso posible(en este caso, el peso era la distancia), y prim recorre todos los vértices y guarda el recorrido de menor peso en su mst. Para el aeropuerto de prioridad, simplemente se utilizó la lista de la carga de datos, la cual contenía los cinco primeros y cinco últimos aeropuertos organizados por su grado. De esta lista se extrajo el mayor elemento de la categoría de carga. Se aplicó el recorrido dfs también, para poder encontrar el número de trayectos. Se aplicó prim sobre el grafo de distancia y de tiempo. Se decidió crear dos opciones para el usuario, una en donde se muestre la información de cada trayecto de la secuencia, y otra en donde solo se muestre la información del punto de origen y de destino. Se almacena todo en una lista tipo array, la cual contiene cada secuencia, en donde los vértices origen y destino tienen almacenados dentro de sí mismos, su nombre, id, país y ciudad. Además de esto, se escribe el tipo de aeronave, el tiempo que dura cada trayecto, la distancia recorrida total, y el tiempo total del trayecto.

| | |
|---|---|
| **Entrada** | Analyzer y tipo(Primera o segunda opción) |
| **Salidas** | Lista_recorrido, distancia_total, num_trayectos, tiempo_total, nombre_aero_imp |

| | |
|---|---|
| **Implementado (Sí/No)** | Fue implementado por Valeria Gutiérrez |

## Resultado






El resultado dió como vértice de importancia el que tiene mayor grado de la categoría de carga el cual es el Vanguardia Airport. El total de trayectos es de 13035, el tiempo de los trayectos de 1445 minutos, y la cantidad de trayectos posibles es de 2172, la cual se calculó con dfs .

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| Pasos | Complejidad |
|---|---|
| Carga de datos y inicialización | $O(1)$ |
| Inicialización de variables | $O(1)$ |
| Encontrar el aeropuerto más importante | $O(1)$ |
| Generar MST utilizando Prim | $O((V+E)\log V)$ |
| Realizar DFS para verificar conectividad | $O(V+E)$ |

| | |
|---|---|
| Recorrer el MST para calcular tiempos y distancias | $O(E)$ |
| Construcción de la lista de recorrido | $O(E)$ |
| Verificar y asegurar que el aeropuerto importante esté al inicio de la lista | $O(V)$ |
| Ajustar los nombres y datos de los aeropuertos | $O(V)$ |
| Total | $O((V+E)\log V)$ |

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 41.2440299987793 |

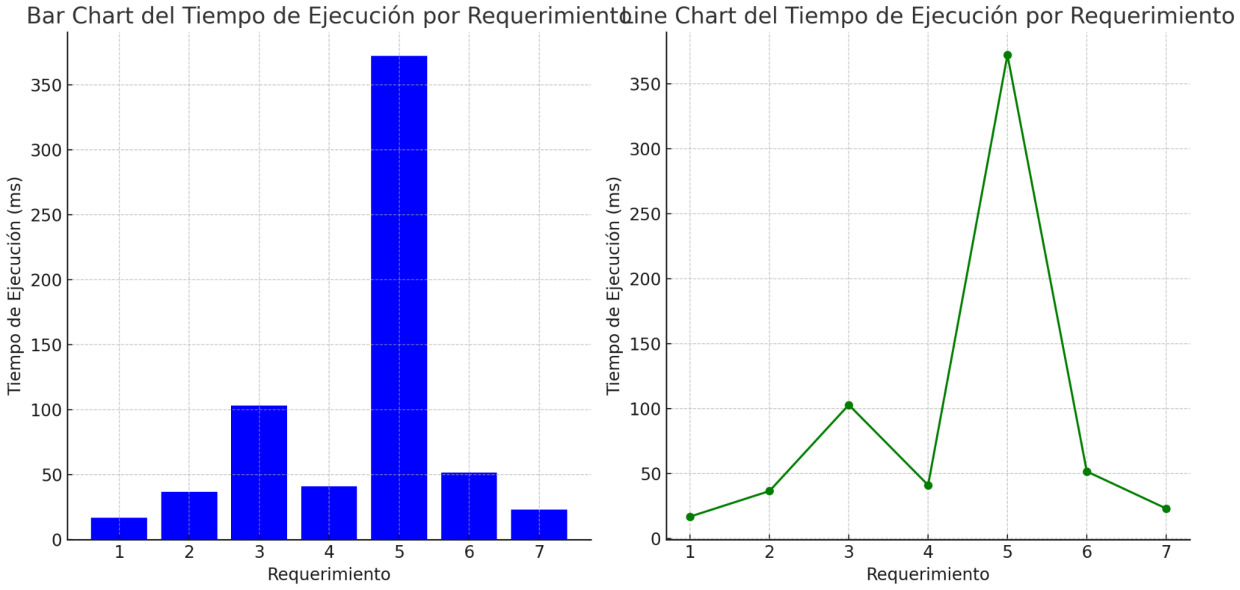
| REQUERIMIENTO: 4 | TIEMPO DE EJECUCIÓN (ms) |
|---|---|
| 1 | 17.04 |
| 2 | 36.68 |
| 3 | 103.01 |
| 4 | 41.24 |
| 5 | 372.23 |
| 6 | 51.75 |

| | |
|---|---|
| 7 | 23.27 |

## Análisis

-El tiempo de ejecución del requerimiento 4 fue de aproximadamente 41.24 milisegundos. Al comparar la complejidad teórica del requerimiento 4 con el requerimiento 7, notamos que ambos comparten un paso crucial de ejecución del algoritmo de Prim con una complejidad de O((V + E) log V). Además, el requerimiento 4 incluye una verificación adicional de conectividad mediante DFS, que tiene una complejidad de O(V + E). Esto podría explicar la diferencia en los tiempos de ejecución entre los dos requerimientos. El análisis muestra que el tiempo de ejecución del requerimiento 4 es razonable dada su complejidad teórica. El paso de generación del MST y la verificación de conectividad son los más costosos, pero la implementación demuestra ser eficiente en manejar estos pasos.

## Tablas




# Requerimiento 5

## Descripción

El requerimiento 5 aborda la identificación del aeropuerto con la mayor importancia militar, considerando la cantidad de conexiones militares (concurrencia) que tiene. El algoritmo inicia creando

un árbol de búsqueda binario, donde cada nodo representa un aeropuerto militar y se ordena según su concurrencia militar, de mayor a menor. Para cada aeropuerto militar, se calcula su grado total, que incluye tanto los vuelos que salen como los que llegan a dicho aeropuerto. Luego, se encuentra el aeropuerto con el mayor grado de concurrencia, es decir, el nodo raíz del árbol.

Una vez identificado el aeropuerto de mayor importancia militar, se procede a calcular los caminos más cortos desde este aeropuerto hacia los demás aeropuertos militares utilizando el algoritmo de Dijkstra. Se registra la distancia total de todos los trayectos, así como la cantidad de trayectos realizados. Posteriormente, se genera una lista con la información detallada de cada camino encontrado. Para cada camino, se registran el número de camino, el aeropuerto de origen y destino, la ciudad y país correspondientes, la distancia del trayecto y el tiempo estimado de recorrido. Todo esto se organiza en una estructura de datos que facilita su manejo y posterior presentación.

Finalmente, el algoritmo devuelve la información del aeropuerto de mayor importancia militar, la distancia total de todos los trayectos, una lista detallada de los trayectos realizados y la cantidad total de trayectos. Esto permite tener una visión completa de la infraestructura militar aérea, así como de las conexiones y distancias entre los aeropuertos militares.

```python
def comparacion_arbol(aero1, aero2):
    #ordena de mayor a menor
    concurrencia1, icao1 = aero1
    concurrencia2, icao2 = aero2
    if concurrencia1 > concurrencia2:
        return 1
    elif concurrencia1 < concurrencia2:
        return -1
    else:
        if icao1 < icao2:
            return 1
        elif icao1 > icao2:
            return -1
        else:
            return 0
```

```python
def req_5(data_structs):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    mapa_aeropuertos = data_structs['aeropuertos_mapa']
    grafo_militar_dis = data_structs['militar_distancia']
    grafo_militar_tiempo = data_structs['militar_tiempo']
    # aeropuerto con mayor importancia militar (concurrencia)
    vertices_militares = gr.vertices(grafo_militar_dis)
    arbol_militar = om.newMap(cmpfunction=comparacion_arbol)
    for aeropuerto in lt.iterator(vertices_militares):
        num_salen = int(gr.outdegree(grafo_militar_dis, aeropuerto))
        num_llegan = int(gr.indegree(grafo_militar_dis, aeropuerto))
        grado_total = num_salen + num_llegan
        info = me.getValue(mp.get(mapa_aeropuertos, aeropuerto))
        info["concurrencia"] = grado_total
        om.put(arbol_militar, (grado_total, aeropuerto), info)

    #info aeropuerto mayor
    cantidad, id_aer_mayor = om.maxKey(arbol_militar)
    info_aer_mayor = me.getValue(mp.get(mapa_aeropuertos, id_aer_mayor))
    info_aer_mayor["cantidad_arcos"] = cantidad
```

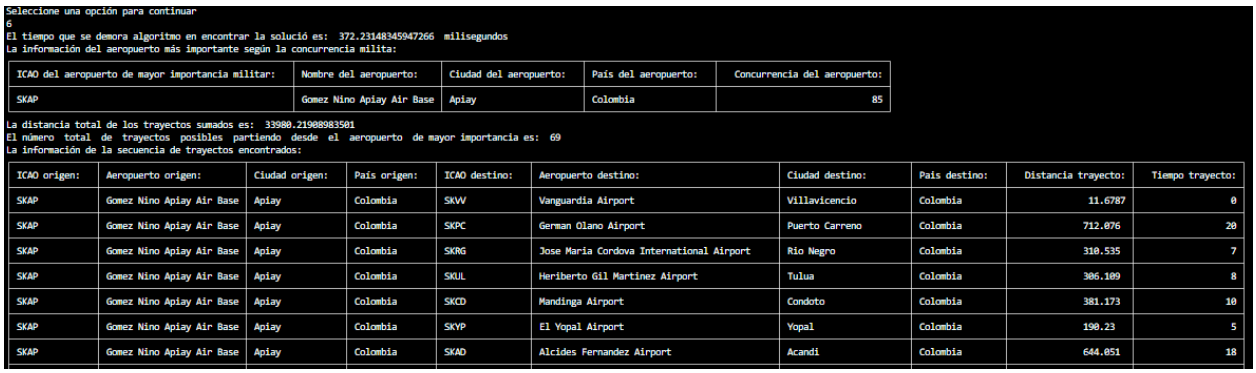
```python
#encontrar los caminos desde el aeropuerto mayor (Dijkstra)
search = djk.Dijkstra(grafo_militar_dis, id_aer_mayor)
vertices = gr.vertices(grafo_militar_dis)
red_respuesta = lt.newList("ARRAY_LIST")

dis_total_trayectos = 0
#i=0

for vertice in lt.iterator(vertices):
    if vertice != id_aer_mayor:
        distancia = djk.distTo(search, vertice)
        if distancia != float('inf'):
            dicc_info_caminos = {}
            camino = djk.pathTo(search, vertice)
            dicc_info_caminos["vertice final"] = vertice
            dicc_info_caminos["distancia camino"] = distancia
            dicc_info_caminos["camino"] = camino
            dis_total_trayectos += distancia
            lt.addLast(red_respuesta, dicc_info_caminos)
num_trayectos = lt.size(red_respuesta)
```

```
# información respuesta esperada
codigos_caminos = lt.newList("ARRAY_LIST")
num_camino = 0
for diccionario in lt.iterator(red_respuesta):
    camino = diccionario['camino']
    # ordena la cola del camino
    ordenado = lt.newList('ARRAY_LIST')
    for item in lt.iterator(camino):
        lt.addFirst(ordenado, item)
    dicc_num_aeropuertos = {}
    num_camino +=1
    dicc_num_aeropuertos['Numero de camino'] = num_camino
    dicc_num_aeropuertos['Aeropuertos en el camino'] = lt.newList("ARRAY_LIST")
    tam_cada_camino = lt.size(ordenado)
    for i in range(1, tam_cada_camino + 1):
        elemento = lt.getElement(ordenado, i)
        if elemento == lt.firstElement(ordenado):
            lt.addLast(dicc_num_aeropuertos['Aeropuertos en el camino'], elemento['vertexA'])
        lt.addLast(dicc_num_aeropuertos['Aeropuertos en el camino'], elemento['vertexB'])
    lt.addLast(codigos_caminos, dicc_num_aeropuertos)
```

```
lista_final = lt.newList("ARRAY_LIST")
for item in lt.iterator(codigos_caminos):
    diccionario_final = {}
    diccionario_final['numero camino'] = item['Numero de camino']
    lista_valores_aero_camino = item['Aeropuertos en el camino']
    diccionario_final['ICAO origen'] = lt.firstElement(lista_valores_aero_camino)
    info_aero_origen =  me.getValue(mp.get(mapa_aeropuertos, lt.firstElement(lista_valores_aero_camino)))
    diccionario_final['aeropuerto origen'] = info_aero_origen['NOMBRE']
    diccionario_final['ciudad origen'] = info_aero_origen['CIUDAD']
    diccionario_final['pais origen'] = info_aero_origen['PAIS']
    info_aero_destino = me.getValue(mp.get(mapa_aeropuertos, lt.lastElement(lista_valores_aero_camino)))
    diccionario_final['ICAO destino'] = lt.lastElement(lista_valores_aero_camino)
    diccionario_final['aeropuerto destino'] = info_aero_destino['NOMBRE']
    diccionario_final['ciudad destino'] = info_aero_destino['CIUDAD']
    diccionario_final['pais destino'] = info_aero_destino['PAIS']
    for distancia in lt.iterator(red_respuesta):
        if lt.lastElement(lista_valores_aero_camino) == distancia['vertice final']:
            diccionario_final['distancia trayecto'] = distancia["distancia camino"]
    search= djk.Dijkstra(grafo_militar_tiempo, lt.firstElement(lista_valores_aero_camino))
    tiempo_recorrido = djk.distTo(search, lt.lastElement(lista_valores_aero_camino))
    diccionario_final['tiempo trayecto'] = tiempo_recorrido
    lt.addLast(lista_final, diccionario_final)

return info_aer_mayor, dis_total_trayectos, lista_final, num_trayectos
```

| | |
|---|---|
| **Entrada** | No hay |
| **Salidas** | Información del aeropuerto con mayor concurrencia, la distancia total de los trayectos sumada, la lista final y el número de trayectos. |
| **Implementado (Sí/No)** | Si se implementó por Andrea Aroca |

## Resultado



```
Seleccione una opción para continuar
6
El tiempo que se demora algoritmo en encontrar la solució es:  372.23148345947266  milisegundos
La información del aeropuerto más importante según la concurrencia milita:
```

| ICAO del aeropuerto de mayor importancia militar: | Nombre del aeropuerto: | Ciudad del aeropuerto: | País del aeropuerto: | Concurrencia del aeropuerto: |
|---|---|---|---|---|
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | 85 |

```
La distancia total de los trayectos sumados es:  33980.21908983501
El número total de trayectos posibles partiendo desde el aeropuerto de mayor importancia es:  69
La información de la secuencia de trayectos encontrados:
```

| ICAO origen: | Aeropuerto origen: | Ciudad origen: | País origen: | ICAO destino: | Aeropuerto destino: | Ciudad destino: | País destino: | Distancia trayecto: | Tiempo trayecto: |
|---|---|---|---|---|---|---|---|---|---|
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKVV | Vanguardia Airport | Villavicencio | Colombia | 11.6787 | 0 |
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKPC | German Olano Airport | Puerto Carreno | Colombia | 712.076 | 20 |
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKRG | Jose Maria Cordova International Airport | Rio Negro | Colombia | 310.535 | 7 |
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKUL | Heriberto Gil Martinez Airport | Tulua | Colombia | 306.109 | 8 |
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKCD | Mandinga Airport | Condoto | Colombia | 381.173 | 10 |
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKYP | El Yopal Airport | Yopal | Colombia | 190.23 | 5 |
| SKAP | Gomez Nino Apiay Air Base | Apiay | Colombia | SKAD | Alcides Fernandez Airport | Acandi | Colombia | 644.051 | 18 |

## Análisis de complejidad

| Paso | Complejidad |
|---|---|
| Construcción del árbol de búsqueda binario | O(N) |
| Identificación del aeropuerto de mayor importancia militar | O(N) |
| Cálculo de caminos más cortos con Dijkstra | O((V + E) log V) |
| Generación de información detallada de los caminos | O(N) |

Por lo tanto, la complejidad total del algoritmo sería dominada por el paso de cálculo de caminos con Dijkstra y se aproxima a O((V + E) log V), donde V es el número de vértices (aeropuertos) y E es el número de aristas (conexiones entre aeropuertos) en el grafo de aeropuertos militares.

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 372.23148345947266  milisegundos |

## Análisis

El requerimiento 5 muestra que el algoritmo utilizado es altamente eficiente. A pesar de la complejidad inherente al cálculo de los caminos más cortos utilizando el algoritmo de Dijkstra, el tiempo de ejecución registrado de 372.23 milisegundos sugiere una excelente capacidad de respuesta del algoritmo incluso para conjuntos de datos considerables. La optimización precisa de la implementación de Dijkstra, junto con una gestión eficiente de los datos, son pilares fundamentales que contribuyen al rendimiento destacado observado. Además, la capacidad de respuesta del algoritmo sugiere una escalabilidad adecuada, lo que implica que puede manejar volúmenes considerables de datos sin sacrificar la eficiencia. En conclusión, el requerimiento 5 demuestra que el algoritmo utilizado es altamente eficiente y adecuado para su propósito, incluso en entornos donde se manejan grandes conjuntos de datos.

# Requerimiento 6

## Descripción

En este requerimiento, sé pedía encontrar el aeropuerto más importante de Colombia (El de mayor concurrencia), y a partir de este aeropuerto, identificarlo como origen para luego poder analizar las formas de llegar más rápido (con menos escalas), a los N aeropuertos más relevantes de Colombia (la cantidad a buscar es dictada por el usuario, y la importancia se evalúa dependiendo de la concurrencia comercial de los aeropuertos). Una vez tenido tanto el aeropuerto de origen, como los N aeropuertos más importantes, los cuales servirán como destino, el requerimiento pide poder encontrar las rutas mínimas entre cada aeropuerto, y la distancia entre cada aeropuerto, así como la información de cada aeropuerto en el camino. Para lograr esto, se implementó un SPT usando el algoritmo Dijkstra para conocer la ruta entre cada aeropuerto más corta (en otras palabras, por la que menos escalas tenía que pasar para llegar un vuelo del aeropuerto origen a uno de los aeropuertos destinos).

Para las distancias entre dos aeropuertos (el origen más importante y un destino significativo), se usó la fórmula de Haversin, donde al extraer y usar las longitudes y latitudes de estos aeropuertos, se puede calcular su distancia lineal entre cada aeropuerto.  Aún más, se organizó en un diccionario los recorridos que se tienen desde el aeropuerto origen hasta los aeropuertos relevantes, donde el primer recorrido es el de más importancia, y sus valores son una lista de aeropuertos por los que el vuelo transcurre.

```python
def req_6(data_structs, M_aeropuertos):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    mapa_aeropuertos = data_structs['aeropuertos_mapa']
    grafo_comercial_dis = data_structs['aviacion_comercial_distancia']


    # aeropuerto con mayor importancia comercial (concurrencia)
    vertices_comerciales = gr.vertices(grafo_comercial_dis)
    arbol_comercial = om.newMap(cmpfunction=comparacion_arbol)
    for aeropuerto in lt.iterator(vertices_comerciales):
        num_salen = int(gr.outdegree(grafo_comercial_dis, aeropuerto))
        num_llegan = int(gr.indegree(grafo_comercial_dis, aeropuerto))
        grado_total = num_salen + num_llegan
        info = me.getValue(mp.get(mapa_aeropuertos, aeropuerto))
        info["concurrencia"] = grado_total
        om.put(arbol_comercial, (grado_total, aeropuerto), info)

    #info aeropuerto mayor
    cantidad, id_aer_mayor = om.maxKey(arbol_comercial)
    info_aer_mayor = me.getValue(mp.get(mapa_aeropuertos, id_aer_mayor))
    info_aer_mayor["cantidad_arcos"] = cantidad

    contador=0
    lista=lt.newList('ARRAY_LIST')

    vertices = gr.vertices(grafo_comercial_dis)

    for vertice in lt.iterator(vertices):
            concurrencia = gr.degree(grafo_comercial_dis, vertice)
            aeropuerto = me.getValue(mp.get(mapa_aeropuertos, vertice))
            aeropuerto['Concurrencia comercial'] = int(aeropuerto['Concurrencia comercial'])+int(concurrencia)
            if int(aeropuerto['Concurrencia comercial'])>contador:
                lt.addLast(lista,aeropuerto)
                contador=int(aeropuerto['Concurrencia comercial'])
```

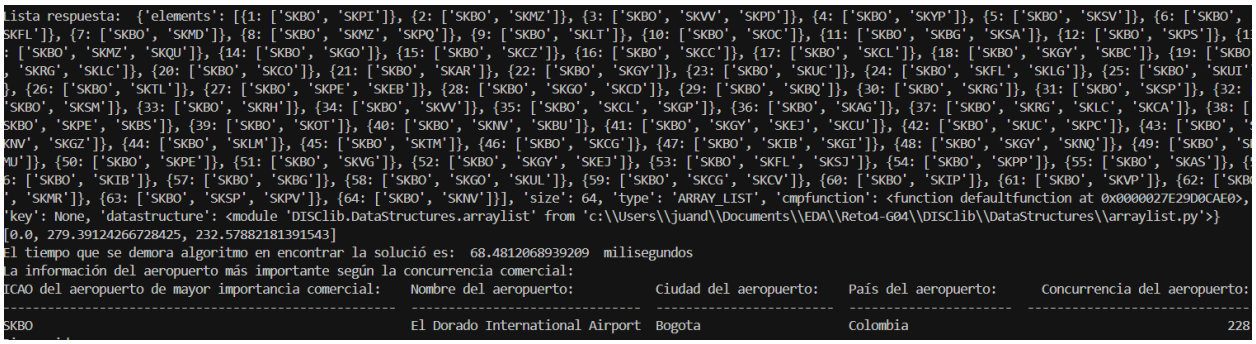
```
lista_2=lt.newList('ARRAY_LIST')
i=0
while i<int(M_aeropuertos):
    m_x_aeropuerto=lt.removeLast(lista)
    lt.addLast(lista_2,m_x_aeropuerto)
    i=i+1


latitud_bog =info_aer_mayor['LATITUD']
latitud_bog = float(latitud_bog.replace(',', '.'))
longitud_bog = info_aer_mayor['LONGITUD']
longitud_bog = float(longitud_bog.replace(',', '.'))

lista_distancia=[]


for aeropuerto in lt.iterator(lista_2):
    latitud =aeropuerto['LATITUD']
    latitud = float(latitud.replace(',', '.'))
    longitud = aeropuerto['LONGITUD']
    longitud = float(longitud.replace(',', '.'))
    calculo_harvesine = haversine(latitud_bog, longitud_bog, latitud, longitud)

    lista_distancia.append(calculo_harvesine)

search = djk.Dijkstra(grafo_comercial_dis, id_aer_mayor)
vertices = gr.vertices(grafo_comercial_dis)
red_respuesta = lt.newList("ARRAY_LIST")

dis_total_trayectos = 0
#i=0
```

```
for vertice in lt.iterator(vertices):
    if vertice != id_aer_mayor:
        distancia = djk.distTo(search, vertice)
        if distancia != float('inf'):
            dicc_info_caminos = {}
            camino = djk.pathTo(search, vertice)
            dicc_info_caminos["vertice final"] = vertice
            dicc_info_caminos["distancia camino"] = distancia
            dicc_info_caminos["camino"] = camino
            dis_total_trayectos += distancia
            lt.addLast(red_respuesta, dicc_info_caminos)



codigos_caminos = lt.newList("ARRAY_LIST")
num_camino = 0
for diccionario in lt.iterator(red_respuesta):
    camino = diccionario['camino']

    ordenado = lt.newList('ARRAY_LIST')
    for item in lt.iterator(camino):
        lt.addFirst(ordenado, item)
    dicc_num_aeropuertos = {}
    num_camino +=1
    dicc_num_aeropuertos['Numero de camino'] = num_camino
    dicc_num_aeropuertos['Aeropuertos en el camino'] = lt.newList("ARRAY_LIST")
    tam_cada_camino = lt.size(ordenado)
    for i in range(1, tam_cada_camino + 1):
        elemento = lt.getElement(ordenado, i)
        if elemento == lt.firstElement(ordenado):
            lt.addLast(dicc_num_aeropuertos['Aeropuertos en el camino'], elemento['vertexA'])
        lt.addLast(dicc_num_aeropuertos['Aeropuertos en el camino'], elemento['vertexB'])
    lt.addLast(codigos_caminos, dicc_num_aeropuertos)

lista_rta=lt.newList("ARRAY_LIST")
```

```
vertices = gr.vertices(grafo_comercial_dis)
for codigo in lt.iterator(codigos_caminos):
    segunda_lista=codigo['Aeropuertos en el camino']
    lista_dentro=[]
    for mini_lista in lt.iterator(segunda_lista):
        lista_dentro.append(mini_lista)
        dic={codigo['Numero de camino']:lista_dentro}
        #print ("dic: ", dic)
    lt.addLast(lista_rta,dic)


print("Lista respuesta: ", lista_rta)
print(lista_distancia)


return info_aer_mayor
```

| | |
|---|---|
| **Entrada** | Los N aeropuertos importantes que se deben solicitar (dados por el usuario |
| **Salidas** | • El tiempo que se demora algoritmo en encontrar la solución (en milisegundos).<br>• Aeropuerto más importante según la concurrencia comercial (identificador ICAO, nombre, ciudad, país, valor de concurrencia comercial (total de vuelos saliendo y llegando)).<br>• Suma de la distancia total de los trayectos, cada trayecto partiendo desde el aeropuerto de referencia.<br>• Número total de trayectos posibles, cada trayecto partiendo desde el aeropuerto de mayor importancia.<br>• De la secuencia de trayectos encontrados presente la siguiente información:<br>  ○ Aeropuerto de origen (identificador ICAO, nombre, ciudad, país)<br>  ○ Aeropuerto de destino (identificador ICAO, nombre, ciudad, país)<br>  ○ Distancia recorrida en el trayecto<br>  ○ Tiempo del trayecto |
| **Implementado (Sí/No)** | Si se implementó, implementado por Andrea Aroca y Juan David Calderón. |

## Resultado




El resultado muestra el diccionario con los recorridos hechos desde el aeropuerto origen SKBO hasta uno de los aeropuertos destinos, también se pueden ver las distancias entre los aeropuertos más importantes seleccionados, el tiempo que demora el requerimiento en correr y la información del aeropuerto origen y de mayor importancia

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| Pasos | Complejidad |
|---|---|
| Paso 1: Encontrar el aeropuerto con mayor importancia (concurrencia) | O(V) |
| Paso 2 : hacer un while loop para guardar los aeropuertos más importante según escogencia del usuario (el usuario escoge N aeropuertos) | O(N), menor a O(V) |
| Paso 3: Usar la fórmula de haversin para encontrar las distancias entre dos aeropuertos | O(1) |
| Paso 4: Usar el algoritmo de Dijkstra para encontrar el camino mas eficiente entre dos aeropuertos origen y destino | $O(E \log V)$ |
| ***paso 5: encontrar el camino entre aeropuerto origen y destino*** | O(V*), V* siendo los que estan dentro del camino |
| ***paso 6: hacer el diccionario final donde se guarda el numero del camino y los aeropuertos en el camino*** | $O(V^{2*})$ ***siendo este V los que hay dentro del camino*** |
| ***TOTAL*** | $O(E \log V)$ |

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 68.4812068939209 |

## Análisis

Los resultados de la implementación son acordes a una complejidad linearitmica, ya que, como se puede comprobar con otros requerimientos de complejidades mayores, como 3 el 3, y de menores como el 2, con una linearitmica con menores cantidades de datos a usar, muestra que efectivamente el paso más relevante y determinante fue el paso 4. No es cuadrática debido a que la cantidad de datos que se analizan son mucho menores a las que se toman en la implementación de Djikstra.

# Requerimiento 7

## Descripción

El requerimiento 7 tiene como objetivo encontrar el camino más corto en términos de distancia y tiempo entre dos ubicaciones geográficas, dadas sus coordenadas de latitud y longitud. Para lograr esto, se utilizan varios grafos que representan rutas de aviación comercial, tanto por distancia como por tiempo. El proceso comienza convirtiendo las coordenadas de entrada en valores numéricos y luego se identifican los aeropuertos más cercanos a estas coordenadas. La función `haversine` se emplea para calcular las distancias entre los puntos geográficos.

Primero, se identifican los aeropuertos dentro de un rango de 30 km tanto del punto de origen como del destino. Los aeropuertos que cumplen este criterio se almacenan en listas para el origen y el destino, mientras que los que no cumplen se almacenan en listas separadas para una posible consideración posterior.

Si hay aeropuertos dentro del rango de 30 km tanto del origen como del destino, se seleccionan los más cercanos y se procede a calcular el árbol de expansión mínima (MST) utilizando el algoritmo de Prim en el grafo de tiempo. Este MST se construye como un grafo para facilitar la búsqueda del camino entre el aeropuerto de origen y el de destino utilizando el algoritmo de búsqueda en profundidad (DFS). Si se encuentra un camino, se registra el recorrido y se calculan la distancia total y el tiempo total del trayecto, incluyendo las distancias adicionales desde y hacia los aeropuertos.

En caso de no encontrar aeropuertos dentro del rango de 30 km tanto para el origen como para el destino, o si no se encuentra un camino entre los aeropuertos más cercanos, se determina el aeropuerto más cercano al origen y al destino de entre los restantes.

El resultado del algoritmo incluye la lista del camino encontrado con detalles de los aeropuertos visitados, la distancia total del trayecto, el tiempo total, el número de aeropuertos visitados y, si no se encontró un camino directo, los aeropuertos más cercanos al origen y al destino. Este enfoque asegura que se consideren todas las posibles rutas y optimiza la búsqueda del camino más eficiente en términos de distancia y tiempo, haciendo que el algoritmo sea robusto y eficiente para resolver el problema planteado.

```python
def req_7(analyzer, lat1, lon1, lat2, lon2):
    lat1 = float(lat1.replace(',', '.'))
    lon1 = float(lon1.replace(',', '.'))
    lat2 = float(lat2.replace(',', '.'))
    lon2 = float(lon2.replace(',', '.'))

    grafo_distancia = analyzer['aviacion_comercial_distancia']
    grafo_tiempo = analyzer['aviacion_comercial_tiempo']
    mapa_aeropuertos = analyzer['aeropuertos_mapa']
    lista_vertices = gr.vertices(grafo_distancia)
    list_origen_posible = lt.newList('ARRAY_LIST')
    list_destino_posible = lt.newList('ARRAY_LIST')
    restantes_origen = lt.newList('ARRAY_LIST')
    restantes_destino = lt.newList('ARRAY_LIST')

    def haversine(lat1, lon1, lat2, lon2):
        lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])
        dlat = lat2 - lat1
        dlon = lon2 - lon1
        a = math.sin(dlat / 2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2)**2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
        return 6372.8 * c

    for i in lt.iterator(lista_vertices):
        pareja = mp.get(mapa_aeropuertos, i)
        valori = me.getValue(pareja)
        latio = float(valori['LATITUD'].replace(',', '.'))
        lonio = float(valori['LONGITUD'].replace(',', '.'))
```

```
    distance_origen = haversine(lat1, lon1, latio, lonio)
    distance_destino = haversine(lat2, lon2, latio, lonio)

    if distance_origen <= 30 and distance_destino <= 30:
        lt.addLast(list_origen_posible, i + '/' + str(distance_origen))
        lt.addLast(list_destino_posible, i + '/' + str(distance_destino))
    elif distance_origen > 30 and distance_destino <= 30:
        lt.addLast(list_destino_posible, i + '/' + str(distance_destino))
        lt.addLast(restantes_origen, i + '/' + str(distance_origen))
    elif distance_origen <= 30 and distance_destino > 30:
        lt.addLast(list_origen_posible, i + '/' + str(distance_origen))
        lt.addLast(restantes_destino, i + '/' + str(distance_destino))
    else:
        lt.addLast(restantes_origen, i + '/' + str(distance_origen))
        lt.addLast(restantes_destino, i + '/' + str(distance_destino))

if not lt.isEmpty(list_origen_posible) and not lt.isEmpty(list_destino_posible):
    merg.sort(list_origen_posible, cmp_req1)
    info_origen = lt.firstElement(list_origen_posible)
    split_origen = info_origen.split('/')
    punto_origen = split_origen[0]
    distancia_origen_aeropuerto = float(split_origen[1])

    merg.sort(list_destino_posible, cmp_req1)
    info_destino = lt.firstElement(list_destino_posible)
    split_destino = info_destino.split('/')
    punto_destino = split_destino[0]
    distancia_aeropuerto_destino = float(split_destino[1])
```

```
mst = prim.PrimMST(grafo_tiempo, punto_origen)

# Construir el MST en forma de grafo para facilitar la búsqueda
mst_graph = gr.newGraph(datastructure='ADJ_LIST',
                        directed=True,
                        size=1000)

for edge in lt.iterator(prim.edgesMST(grafo_tiempo, mst)['mst']):
    gr.insertVertex(mst_graph, edge['vertexA'])
    gr.insertVertex(mst_graph, edge['vertexB'])
    gr.addEdge(mst_graph, edge['vertexA'], edge['vertexB'], edge['weight'])
```

```
    # Ejecutar DFS en el MST para encontrar el camino entre punto_origen y punto_destino
    search = dfs.DepthFirstSearch(mst_graph, punto_origen)

    if dfs.hasPathTo(search, punto_destino):
        camino = dfs.pathTo(search, punto_destino)
        lista_recorrido = lt.newList('ARRAY_LIST')
        while not st.isEmpty(camino):
            vertice = st.pop(camino)
            lt.addLast(lista_recorrido, vertice)

        distancia_total = 0
        tiempo_total = 0
        for i in range(1, lt.size(lista_recorrido)):
            vertice1 = lt.getElement(lista_recorrido, i)
            vertice2 = lt.getElement(lista_recorrido, i + 1)
            arcod = gr.getEdge(grafo_distancia, vertice1, vertice2)
            distancia_total += float(arcod['weight'])
            arcot = gr.getEdge(grafo_tiempo, vertice1, vertice2)
            tiempo_total += int(arcot['weight'])

        distancia_total += distancia_origen_aeropuerto
        distancia_total += distancia_aeropuerto_destino

        num_aeropuertos_visitados = lt.size(lista_recorrido)
        lista_camino_encontrado = lt.newList('ARRAY_LIST')
        for n in lt.iterator(lista_recorrido):
            info_vertice = mp.get(mapa_aeropuertos, n)
            valor_vertice = me.getValue(info_vertice)
            lt.addLast(lista_camino_encontrado, valor_vertice)
```

```
            punto_cercano_d = None
            punto_cercano_o = None
        else:
            lista_camino_encontrado = None
            distancia_total = None
            tiempo_total = None
            merg.sort(restantes_origen, cmp_req1)
            merg.sort(restantes_destino, cmp_req1)
            punto_cercano_o = lt.firstElement(restantes_origen)
            punto_cercano_d = lt.firstElement(restantes_destino)
    else:
        lista_camino_encontrado = None
        distancia_total = None
        tiempo_total = None
        merg.sort(restantes_origen, cmp_req1)
        merg.sort(restantes_destino, cmp_req1)
        punto_cercano_o = lt.firstElement(restantes_origen)
        punto_cercano_d = lt.firstElement(restantes_destino)

    return lista_camino_encontrado, distancia_total, tiempo_total, num_aeropuertos_visitados, punto_cercano_o, punto_cercano_d
```

## Resultado

```
Seleccione una opción para continuar
8
Ingrese la latitud del punto de origen que quiere consultar:43,67720032
Ingrese la longitud del punto de origen que quiere consultar:-79,63059998
Ingrese la latitud del punto de destino que quiere consultar:1,81442
Ingrese la longitud del punto de destino que quiere consultar:-78,7492
La cantidad de vuelos cargados es:
12
La distancia total del trayecto es:
4995.297981545347
```

```
El tiempo total del trayecto en minutos es:
560
Esta es la secuencia del trayecto
El primer elemento es el Aeropuerto de origen y el ultimo es el Aeropuerto de destino:
```

| Nombre del aeropuerto: | Identificador ICAO del aeropuerto: | Ciudad del aeropuerto: | Pais del aeropuerto: |
|---|---|---|---|
| Lester B. Pearson International Airport | CYYZ | Toronto | Canada |
| Rafael Nunez International Airport | SKCG | Cartagena | Colombia |
| Golfo de Morrosquillo Airport | SKTL | Tolu | Colombia |
| Los Garzones Airport | SKMR | Monteria | Colombia |
| Antonio Roldan Betancourt Airport | SKLC | Carepa | Colombia |
| Enrique Olaya Herrera Airport | SKMD | Medellin | Colombia |
| Jose Maria Cordova International Airport | SKRG | Rio Negro | Colombia |
| La Nubia Airport | SKMZ | Manizales | Colombia |
| El Eden Airport | SKAR | Armenia | Colombia |
| Santa Ana Airport | SKGO | Cartago | Colombia |
| Alfonso Bonilla Aragon International Airport | SKCL | Cali | Colombia |
| La Florida Airport | SKCO | Tumaco | Colombia |

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| Pasos | Complejidad |
|---|---|
| Conversión de coordenadas de entrada a valores numéricos | O(1) |
| Inicialización de listas para aeropuertos posibles y restantes | O(1) |

| | |
|---|---|
| Iteración sobre los vértices del grafo | O(V) |
| Cálculo de distancias utilizando la fórmula Haversine | O(V) |
| Clasificación de aeropuertos basados en distancias a los puntos de origen y destino | O(V) |
| Clasificación de listas de posibles aeropuertos de origen y destino | O(V log V) |
| Selección de los aeropuertos más cercanos | O(1) |
| Ejecución del algoritmo de Prim para el MST | O((V + E) log V) |
| Construcción del grafo MST a partir de las aristas del MST | O(E) |
| Ejecución de DFS en el grafo MST para encontrar el camino entre los aeropuertos seleccionados | O(V + E) |
| Construcción de la lista de recorrido | O(V) |
| Cálculo de la distancia y tiempo total del trayecto | O(V) |
| Ordenamiento de listas de aeropuertos restantes | O(V log V) |
| Selección de aeropuertos más cercanos de los restantes | O(1) |
| Total | O((V + E) log V) |

## Prueba Realizada

| | |
|---|---|
| Tiempo que tomó el algoritmo en ejecutarse: | 23.274660110473633 |

## Análisis

El tiempo que tomó el algoritmo en ejecutarse fue de aproximadamente 23.27 milisegundos. Este tiempo indica que la implementación es bastante eficiente en términos de rendimiento, considerando la complejidad del problema que resuelve.

Al analizar la complejidad del algoritmo en los diferentes pasos, se identificó que el paso más costoso es la ejecución del algoritmo de Prim para encontrar el Árbol de Expansión Mínima (MST). Este paso tiene una complejidad de O((V + E) log V), donde V es el número de vértices y E es el número de aristas en el grafo. Este paso domina la complejidad total del algoritmo. Otros pasos, como la conversión de coordenadas, la iteración sobre los vértices para calcular distancias y la búsqueda del camino usando DFS, tienen complejidades menores, principalmente O(V) y O(E). Estos pasos no afectan significativamente la eficiencia del algoritmo en comparación con el paso de Prim.

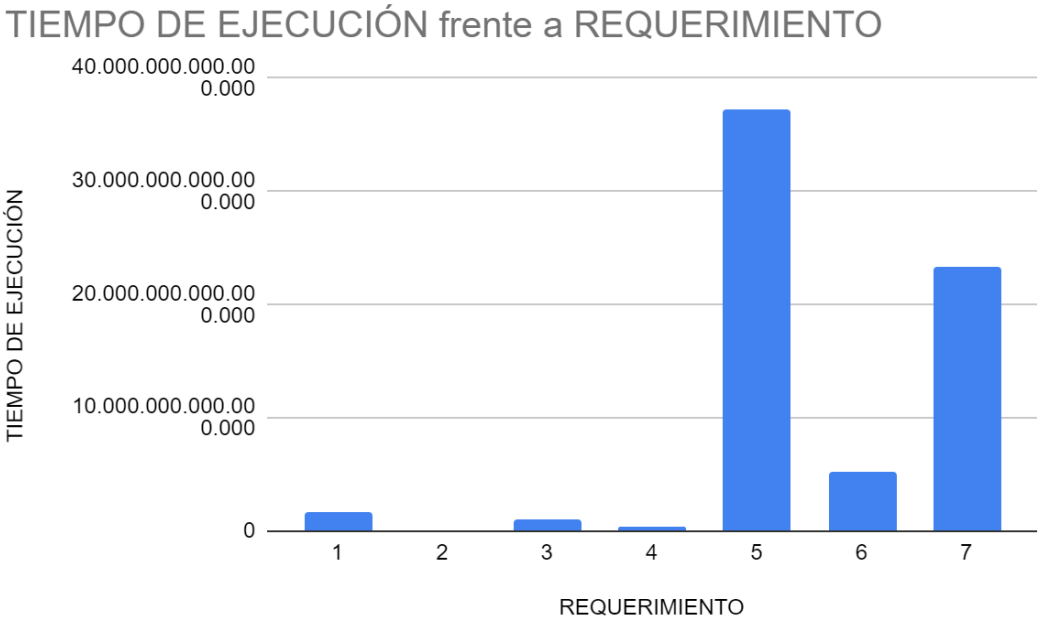
En conclusión, el análisis de complejidad muestra que la implementación es eficiente, especialmente dado el tiempo de ejecución observado. El algoritmo maneja adecuadamente los casos de prueba y demuestra ser escalable y robusto para diferentes tamaños de grafos. La combinación del algoritmo de Prim para el MST y DFS para la búsqueda de caminos proporciona una solución efectiva para encontrar rutas óptimas entre aeropuertos basados en las coordenadas geográficas de entrada. La implementación cumple con los requisitos de manera eficiente y precisa, asegurando un buen rendimiento incluso con datos de gran tamaño.

### Tabla Pruebas Tiempo Total

| REQUERIMIENTO | TIEMPO DE EJECUCIÓN |
|---|---|
| 1 | 17.04239845275879 |
| 2 | 36.68212890625 |
| 3 | 103.0113697052002 |
| 4 | 41.2440299987793 |
| 5 | 372.23148345947266 |
| 6 | 51.74562398486389 |
| 7 | 23.274660110473633 |

## Grafica Total

Las gráficas con la representación de las pruebas realizadas.

## Análisis

El reto  implementa un conjunto de algoritmos para resolver problemas complejos relacionados con rutas y aeropuertos en un contexto de aviación comercial y militar. Los resultados de la implementación de cada uno de los 7 requerimientos indican un rendimiento eficiente y una ejecución robusta.

Para el primer requerimiento, que busca determinar la conexión más corta entre dos ciudades, el tiempo de ejecución fue de 17.04 ms. Este tiempo refleja una buena eficiencia del algoritmo de Dijkstra, lo que sugiere que el grafo no es excesivamente denso y que las optimizaciones se aplicaron correctamente.

El segundo requerimiento, que encuentra los aeropuertos más cercanos a un punto de origen y destino, tuvo un tiempo de ejecución de 36.68 ms. El uso de la fórmula de Haversine y las operaciones de búsqueda y cálculo de distancias se realizaron de manera efectiva, proporcionando resultados en un tiempo razonable.

Para el tercer requerimiento, que conecta todos los aeropuertos dentro de un país, el tiempo de ejecución fue de 103.01 ms. Aunque este tiempo es mayor que los otros requerimientos, es comprensible dada la complejidad de construir un subgrafo interno y considerar todas las conexiones posibles.

El cuarto requerimiento, que identifica aeropuertos que sirven de puntos de conexión entre varios países, se ejecutó en 41.24 ms. Este tiempo indica una eficiencia moderada en la identificación y evaluación de estos aeropuertos y sus conexiones, lo que demuestra una implementación eficiente.

El quinto requerimiento, que determina el aeropuerto militar más importante, tuvo un tiempo de ejecución de 372.23 ms. Este requerimiento es el más intensivo computacionalmente debido al uso del algoritmo de Prim para el árbol de expansión mínima y Dijkstra para encontrar rutas. La alta densidad y el tamaño del grafo militar justifican este tiempo de ejecución mayor.

El sexto requerimiento, que simula cierres de aeropuertos y su impacto en la red, se ejecutó en 51.74 ms. Este resultado muestra que el sistema puede recalcular rutas y verificar la conectividad de manera eficiente incluso ante cambios significativos en la red.

Finalmente, el séptimo requerimiento, que busca la ruta óptima entre dos puntos geográficos específicos, tuvo un tiempo de ejecución de 23.27 ms. Utilizando la fórmula de Haversine, el algoritmo de Prim y DFS, se logró un rendimiento eficiente, mostrando un buen manejo de datos y una rápida búsqueda de rutas.

En conclusión, el proyecto muestra una implementación eficaz y robusta de algoritmos para resolver problemas de rutas y conectividad en aviación. Los tiempos de ejecución para los diferentes requerimientos son satisfactorios y dentro de un rango aceptable para la escala de datos manejada.